ReserveWell
Architecture Notebook

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 system will be deployed to a cloud environment with reliable infrastructure and scalability capabilities.
- The development team possesses expertise in Next.js, React, Firebase, and no-SQL databases.
- The system will handle a moderate volume of traffic during peak hours, with occasional spikes in demand.
- The system will adhere to industry standards 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.

4. Architecturally significant requirements

The following are the architecturally significant requirements for the Restaurant Reservation Management System:

- The system must allow diners to make reservations for restaurants.
- The system must allow diners to manage their reservations.
- The system must allow restaurant staff to view and manage reservations.
- The system must send confirmation emails to diners when they make reservations.
- The system must send notifications to restaurant staff when new reservations are made.

5. Decisions, constraints, and justifications

Decisions:

1. Next.js and React for Front-end Development:
    - Justification: 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. Firebase for Back-end Development:
    - Justification: 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.
3. No-SQL Database for Reservation Data:
    - Justification: 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.

Constraints:

1. Cloud Environment Deployment:

    - Constraint: The system must be designed to be compatible and functional within a cloud environment, adhering to its specific protocols and limitations. Firebase supplies these functionalities.
2. Development Team Expertise:
    - Constraint: The development team's expertise in Next.js, React, Firebase, and no-sql databases is crucial for efficient and successful implementation. Developers must leverage their knowledge to make informed decisions and optimize system performance.
3. Scalability for Traffic Spikes:
    - Constraint: The system must be able to handle occasional spikes in traffic without compromising performance or user experience. Developers must consider load balancing, caching, and autoscaling mechanisms to ensure system resilience.
4. Data Security and Privacy Compliance:
    - Constraint: The system must adhere to industry standards and regulations, such as GDPR and CCPA, to protect user data and privacy. Developers must implement robust data encryption, access controls, and auditing mechanisms.

DOs and DON'Ts:

DOs:

- Utilize cloud-based services to maximize scalability and reduce infrastructure management overhead.
- Employ microservices architecture to promote modularity, independent development, and enhanced fault tolerance.
- Implement caching mechanisms to improve performance and reduce database load.
- Employ load balancing techniques to distribute traffic effectively and prevent bottlenecks.
- Conduct regular security audits and vulnerability assessments to identify and address potential security risks.

DON'Ts:

- Avoid tight coupling between front-end and back-end components to maintain flexibility and independent development.
- Refrain from excessive data storage in the No-SQL database to optimize storage and retrieval efficiency.
- Avoid handling sensitive payment information within the system; rely on a trusted third-party payment gateway.
- Neglect performance optimization techniques, such as caching and load balancing, as they are critical for handling traffic spikes.
- Overlook data security and privacy considerations, as they are paramount in protecting user data and maintaining compliance.

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.

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.

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

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.

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.

Use Case View

The use case view describes the system's behavior from the perspective of its users, focusing on the specific actions they can perform.

1. Make a Reservation:
    - A user selects a restaurant, date, time, and party size.
    - The system verifies availability and creates a new reservation.
    - A confirmation email is sent to the user.
2. View Reservations:
    - A user logs in and views their existing reservations.
    - Reservation details, including restaurant information, date, time, and party size, are displayed.
3. Update a Reservation:
    - A user selects an existing reservation and makes changes.
    - The system updates the reservation record and sends a confirmation email.
4. Cancel a Reservation:
    - A user cancels an existing reservation.
    - The system removes the reservation and sends a cancellation email.