Software Engineering 2 - Prof. Di Nitto Elisabetta
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano

CLup - Customer Line-up

DD
Design Document

Andrea Franchini(10560276)
Ian Di Dio Lavore (10580652)
Luigi Fusco(10601210)

xx-xx-2020

Contents

1 Introduction

1.1 Purpose

The purpose of this document is to give a more detailed view of the *Customers Line-up* system presented in the RASD, explaining architecture, components, processes and algorithms that will satisfy the RASD requirements.

Because of its technical nature, it's aimed towards the software development team. It also includes instructions regarding the implementation, integration and testing plan.

1.2 Scope

Customers Line-Up (CLup) is a system that allows supermarket managers to regulate the influx of people inside physical stores and reduce the time spent in queue by customers.

The idea of CLup is being more akin to an open-source framework that can be adoped and improved modularly, rather than it being a closed-source product.

In particular, CLup allows customers to search and then reserve a visit to a store, either at a specific time or as soon as possible, and get notifyied, if possible, when it's their turn or if there's been a delay in the schedule.

Additionally, CLup aims to provide:

- access to the service via mobile app or website
- physical alternatives for people that do not have Internet access
- book a visit, notifying customers of any change in the schedule
- restrict the store selection by using filters
- suggest alternative stores and/or time frames
- monitor and dynamically restrict the amount of people allowed in a store
- track the time spent in the store by customers to provide better estimate of waiting times

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- *User* (also *Customer* or *Visitor*): A person that uses the system to shop at a store.
- *Registered User*: A User that has registered an Account within the System.
- *System Manager*: A stakeholder (owner, employee, manager etc.) of the Store chain that can tweak the parameters of the System and access informations and statistics.
- *Account*: A reference to a specific User in the System, that allows to track the User across multiple visits.
- *Reservation* (or *Booking*): Arrangement made between a User and the System in which the System shall grant the User access to Store at the arranged time.
- *Visit*: The time frame in which the User enters the store, shops and exits.
- *Time slot*: The time at which a Customer with a Reservation is expected arrive at the store.
- *Store*: Any physical location (e.g.: building) where it is possible to utilize the System.
- *Totem*: A physical device with a touchscreen display and an attached printer that allows Customers to join the Virtual Queue.
- *Virtual Queue*: the virtual equivalent of a physical queue in front of the store, regulating the access of people by ordering them.
- *Web App*: A web application, consisting of a back-end and a front-end accessible from a web browser.
- *Line*: Synonim for *queue*.

1.3.2 Acronyms

- CLup: Customer Line-up
- RASD: Requirement Analysis and Specification Document
- API: Application Programming Interface
- REST: REpresentational State Transfers
- DB: Database
- DBMS: Database Management System
- GPS: Global Positioning System
- MVC: Model-View-Controller (a design pattern)
- CDN: Content Delivery Network

1.3.3 Abbreviations

- [Gn]: n-goal.
- [Dn]: n-domain assumption.
- [Rn]: n-functional requirement.

1.4 Revision History

1.5 Reference Documents

- Problem Specification Document: “Assignment AY 2020-21.pdf”

1.6 Document Structure

The first chapter gives an introduction of the design document and presents to the reader explanations for most of the acronyms and technical language that they'll encounter later in the document.

The second chapter is about the architecture of the system, explaining the most important components, interfaces, patterns as well as deployment and runtime aspects of the system.

The third chapter explains the connection between the UI presented in the RASD and the components presented in this document.

The fourth chapter maps the requirements that have been defined in the RASD to the design elements defined in this document.

The fifth chapter shows the order in which the subcomponents of the system will be implemented as well as the order in which subcomponents will be integrated and how to test the integration.

2 Architectural Design

2.1 Overview

To ensure high maintainability, scalability and security, the service is structured according to the well-established three-tier architecture. Figure 1 shows how the tiers are divided, and what are the relations between key components of the system.

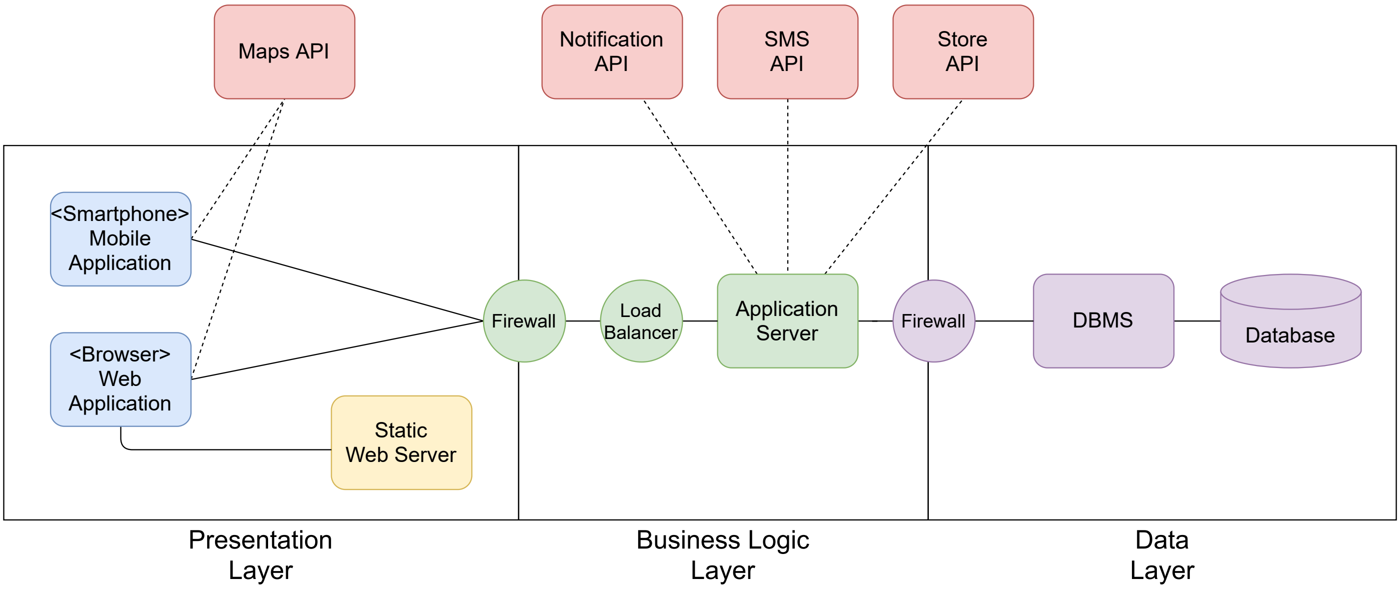

Figure 1: Overall architecture of the System

The main components are the following:

- **Mobile Application** The application is installed on the user's device through its store platform service. The application allows the user to interact with the service and receive notifications from the server.
- **Web Application** The web application allows users to access the same services available on the mobile app through any device, but it's not guaranteed that it can receive notifications. In addition to that, store managers may access a dedicated panel to configure additional parameters.
- **Static Web Server** It serves the client's browser a bundle that contains the web application code (compressed HTML and JS). It has no ties with the application server.
- **Application Server** It's the main backend component of the service, and contains the logic to process requests made against its API from the clients.
- **Database** It's the component that manages the connection to the database.
- **External Services** These services provides functionalities that the service can't provide by itself without additional infrastructure. They incluse a *SMS Service* to send messages to users and a *Maps API* to visualize the location of the store on the user's device.

2.2 Component View

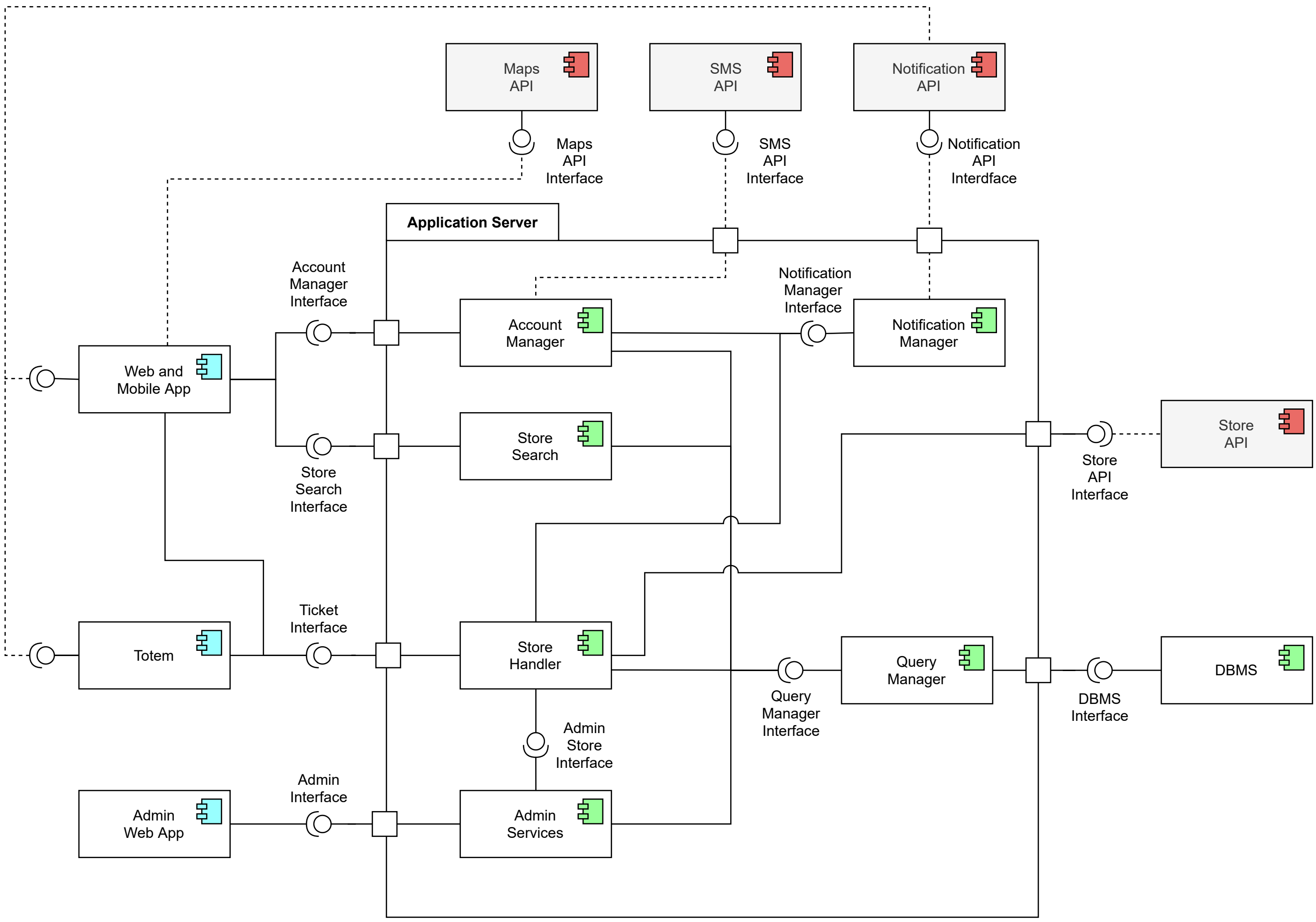

Figure 2: Global Component Diagram of the System

2.2.1 App Components

The application component makes up the entire frontend logic of the system. The role of the application component is to interface the user with the application server API, rendering interfaces and requesting data upon requests. As the entire UI is encoded in the application component, only a minimum amount of data will be passed between frontend and backend, reducind useless and repetitive traffic. The application component is divided in three main subcomponents, each targeted at a different type of user:

- **Web and Mobile App** are targeted at the user. They contain all the logic required to request and display information about stores, reservation, and queues. They are united in a single component as they will share most of the code and will use the same exact API
- **Totem** will be deployed in the totems inside the store. They require a more limited set of functionalities compared to the user components (namely, the possibility of joining the queue). Additionally, it will send request to the Application Server in order to validate tickets. On the UX side, it will contain the functionality needed to print tickets and to display the current status of the queue.
- **Admin Web App** is intended for internal use only. It is the platform through which the store managers will add, remove and manage the stores. It will connect to a different API than the other two components in order to separate the functionalities and the responsibilities as much as possible.

2.2.2 Application Server Components

The Application Server Components contain all the business logic needed to provide the functionalities of the application, communicating with the DBMS when needed and responding to queries sent by the App Components. Its structure is loosely coupled, with only few components relying on the services offered by other components. The Application Server Components are:

Account Manager handles everything related to user accounts. In particular it will offer functionalities related to creating new accounts, logging in, and setting preferences and notifications. When creating an account it will communicate with an external **SMS API** in order to send confirmation codes. add something related to notifications!

Store Search will provide functionalities related to searching stores at specific locations and with filters that will be set by the user

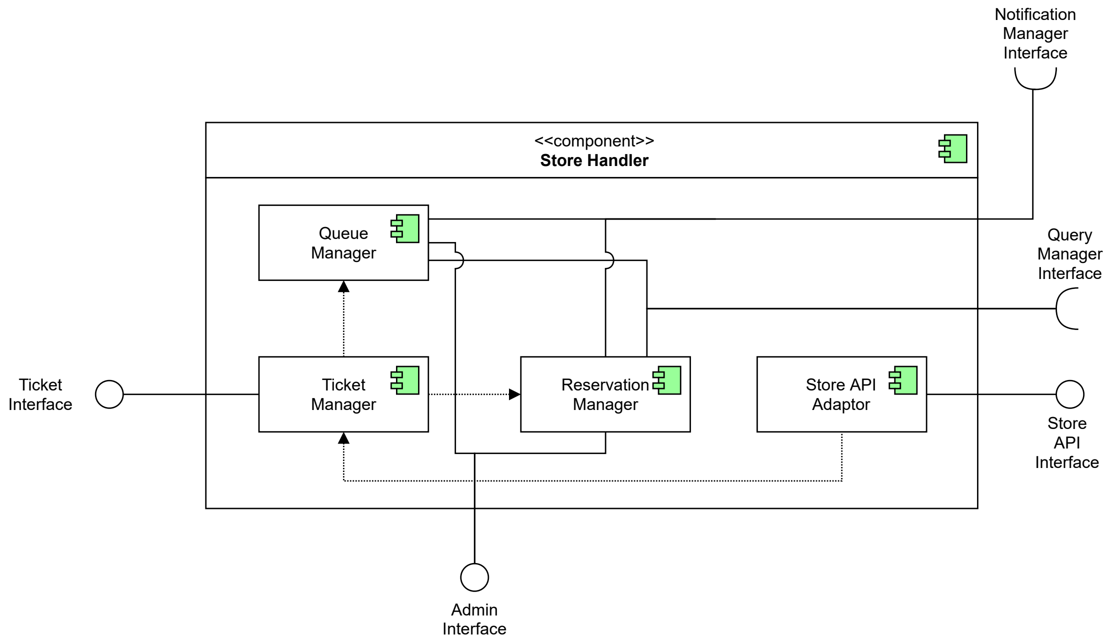

Figure 3: Store Handler Internal View.
The arrows represent a "uses" relationship

Reservation Manager

Queue Manager

Ticket Validator

Account Admin Services

Query Manager

2.2.3 Data Components

The Data layer is composed of a relational database, and its associated DBMS will have the duty of processing and executing parallel requests.

Users and Admins will be stored in different tables. Users can set up Free Timeslots Notifications, in order to be notified when a Timeslot at a specific day in a specific time range is made available for one of the favorite stores. Users have an association with their Tickets, which include both Queue Tickets and Reservation Tickets. In order to preserve the history of Users and for making data analytics possible, Tickets are never deleted, but instead are associated with a status indicating if they are currently active or already used. Each Admin manages a number of Stores, having the power of changing their capacity or all details about their associated Timeslots. Timeslots refer to a specific weekday and have an associated time. In order to keep consistency with Reservation Tickets, Timeslots are immutable and never modified. Their status is instead set to inactive, and another Timeslot is created whenever a change has to be made.

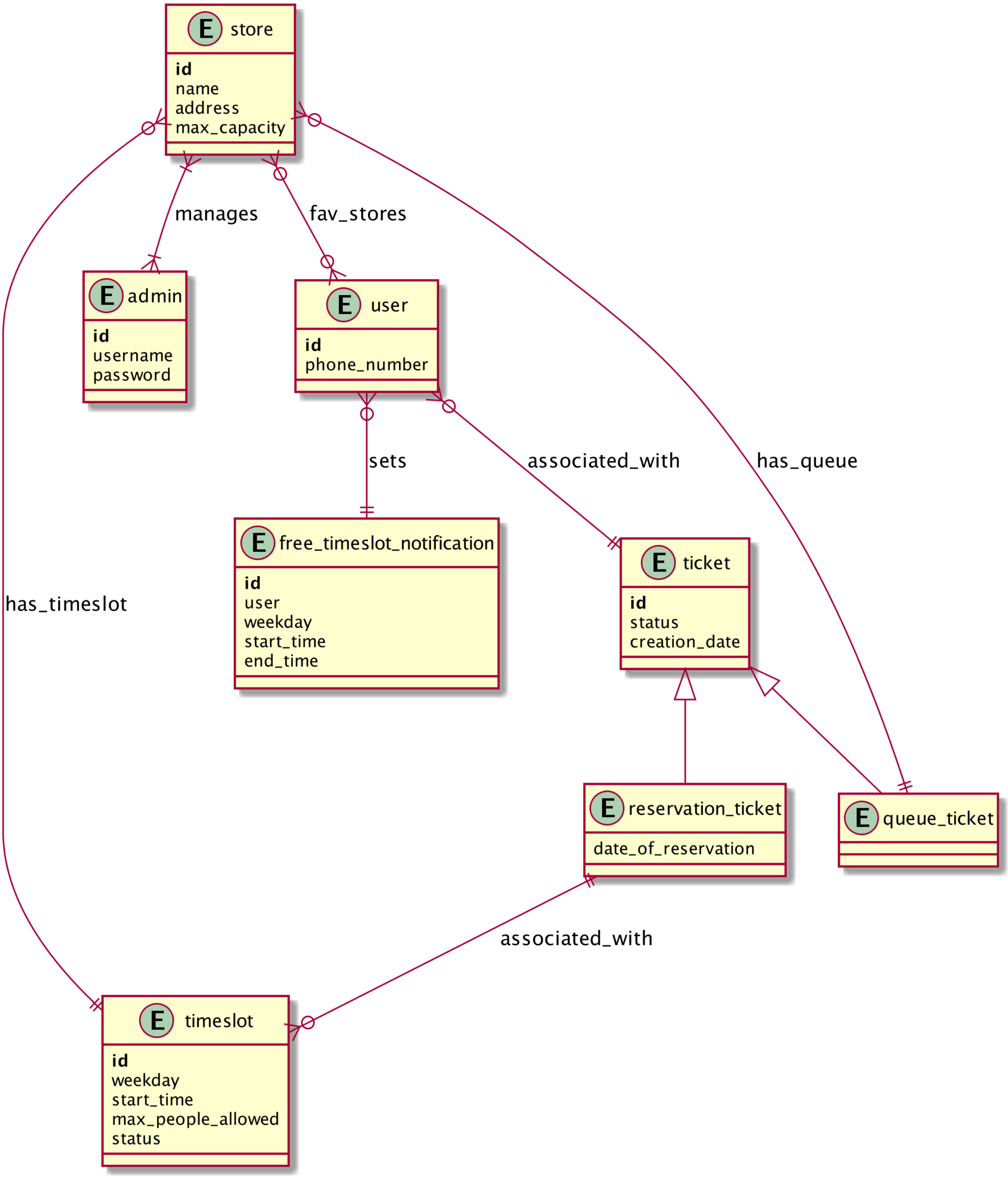

Figure 4: Data Base main structure

2.3 Deployment View

Our system is composed of two independent components: a completely static webserver will be the access point where the client devices will fetch the one-page application, while the application server will offer the APIs to make it work. For this reason we decided to use two different solutions.

The static webserver will make use of Cloudflare's CDN, in order to guarantee immediate response thanks to its edge location caches and reverse proxies. Cloudflare is the obvious choice as they are the major CDN providers in the world

The application server, composed of a business logic and a data tier, will be hosted on a cloud provider, offering many advantages compared to traditional in-house hosting, including:

- **Scalability** thanks to the possibility of allocating new virtual machines, greater per formance cores, or more memory when needed, and to the load balancing services

- **Security** thanks to services like live monitoring and firewalls
- **Cost-Efficiency** as the great flexibility offered by the service allows for paying only the resources that are really needed.

This makes it the ideal service for hosting big and high traffic applications. The chosen cloud provider will have to offer all of the above features.

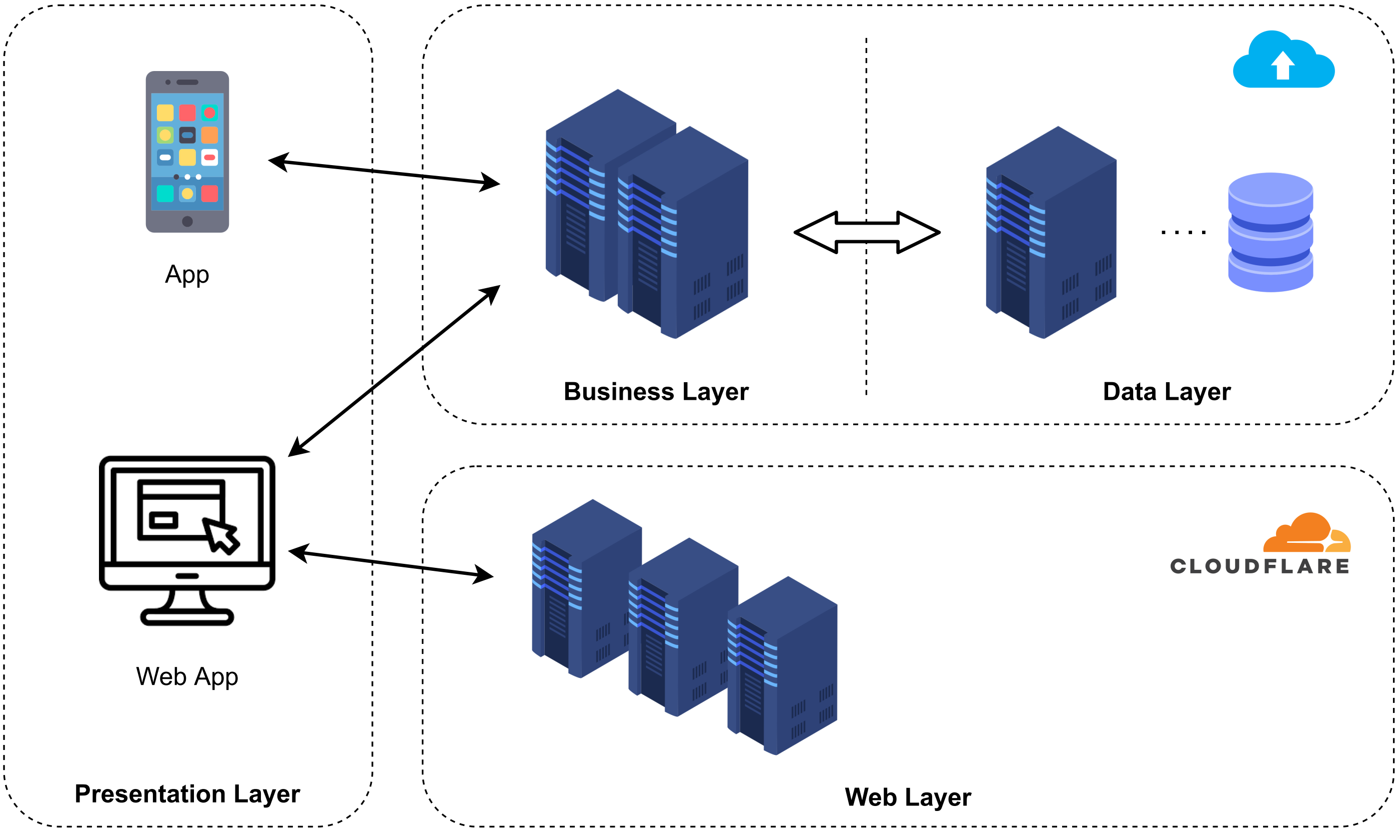

Figure 5: Deployment View

The deployment diagram offers a clearer view over the hardware and software resources of the application:

- **Mobile Device** is any device capable of hosting the mobile application, which has been previously downloaded from an official application store.
- **PC** is any device having a modern browser capable of running the javascript based web app.
- **Cloudflare CDN** will transparently host the one page application, making it available for download without impacting the performance of the main application server. No logic is implemented on this side as the application is completely static, and executes its code on the client machine.
- **Cloud Services** will host the entire business and data logic of the system. It contains:
  - **Firewall** services for filtering incoming connections to the business and data layers
  - **Load Balancer** service for redirecting incoming traffic to the least busy application instance
  - **Application Instances** which will run the business logic in parallel and autonomously, and can be instantiated or deleted when needed
  - **Data Instance** which is a data optimized virtual machine containing the DBMS and the database

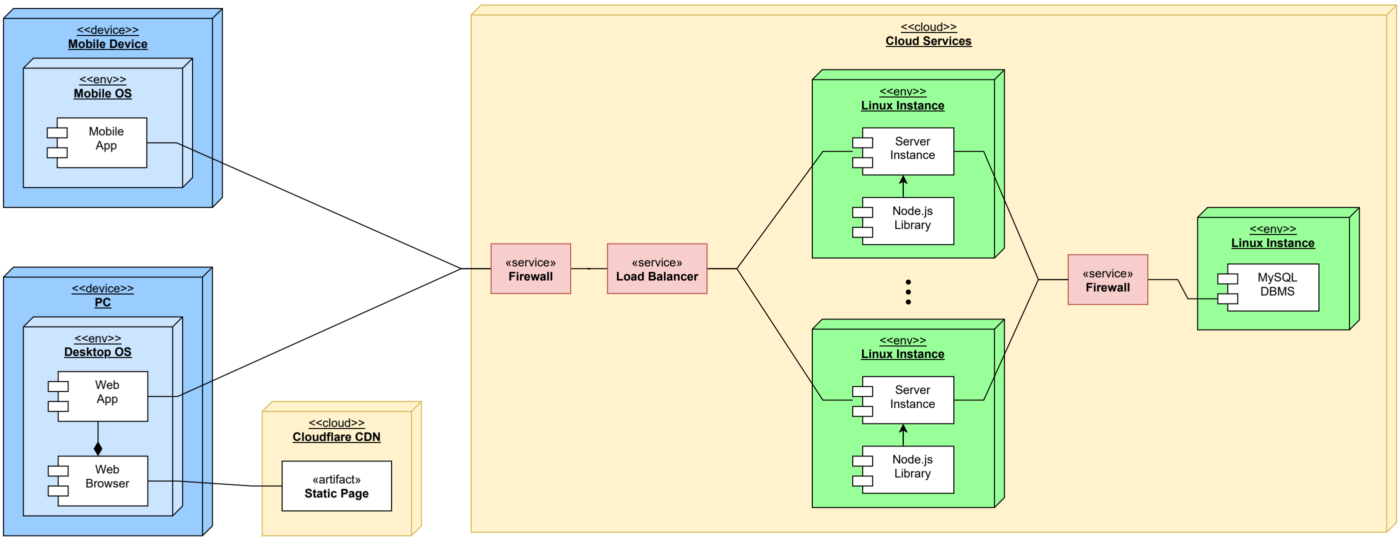

Figure 6: Deployment Diagram

2.4 Runtime View

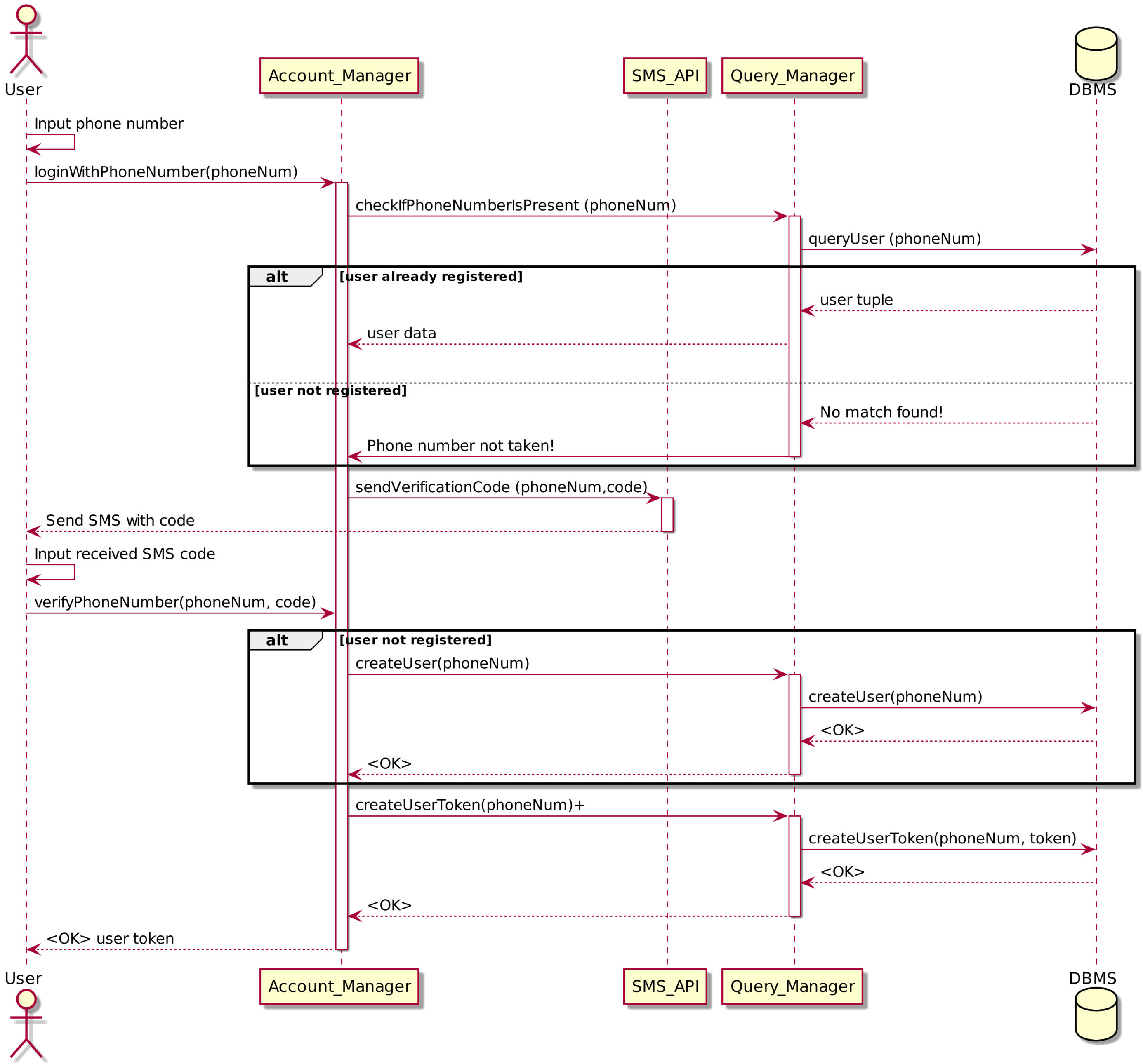

Figure 7: User Login

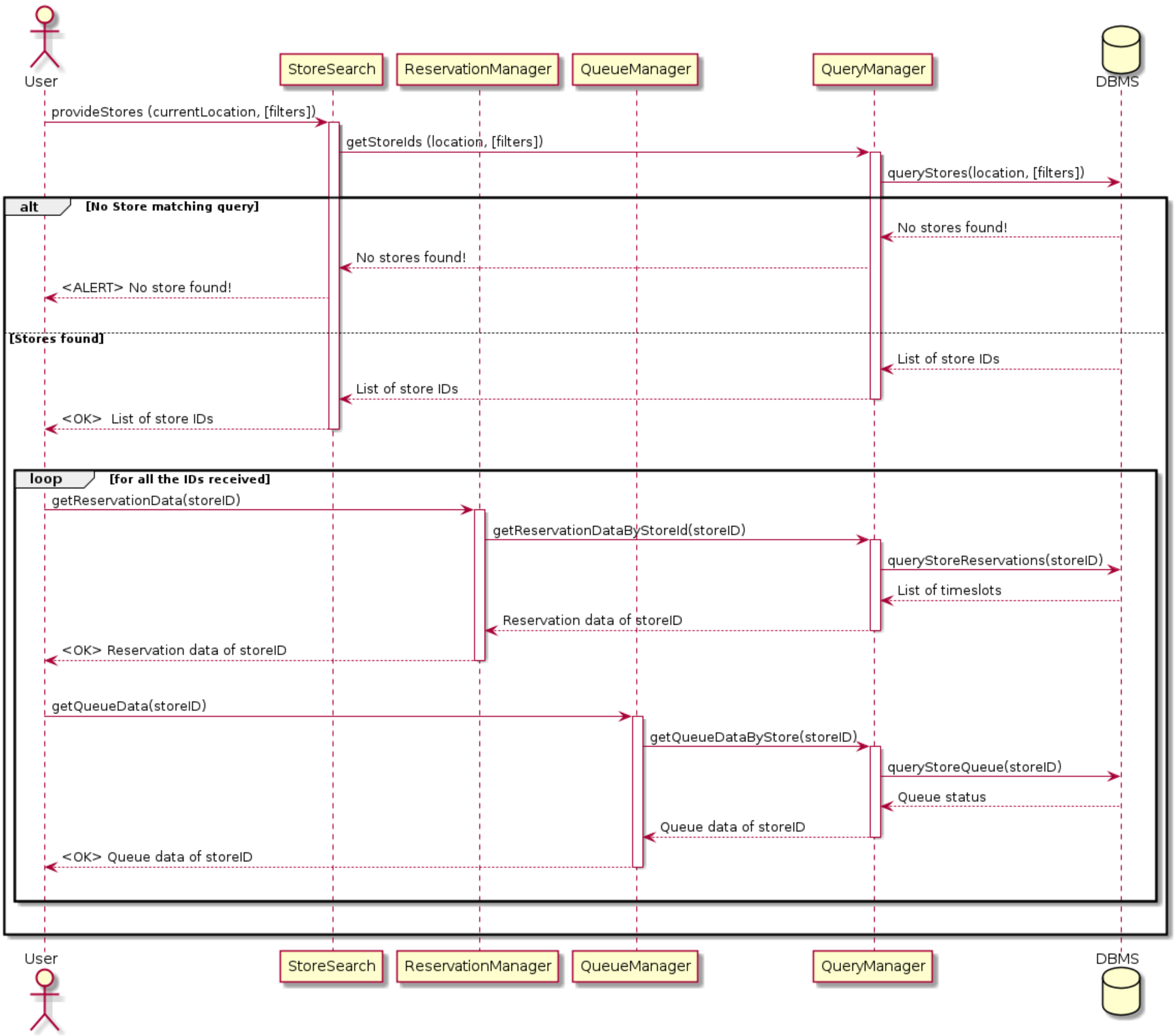

Figure 8: Store search

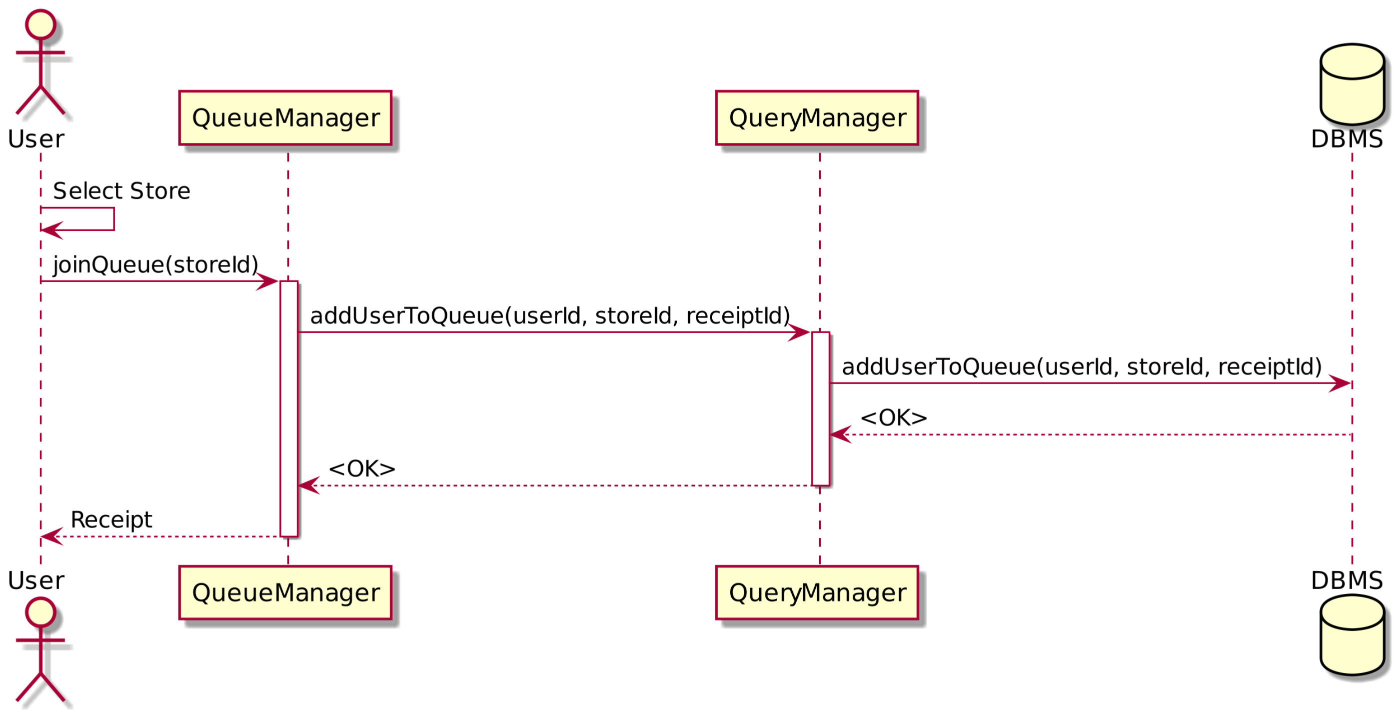

Figure 9: User joins queue

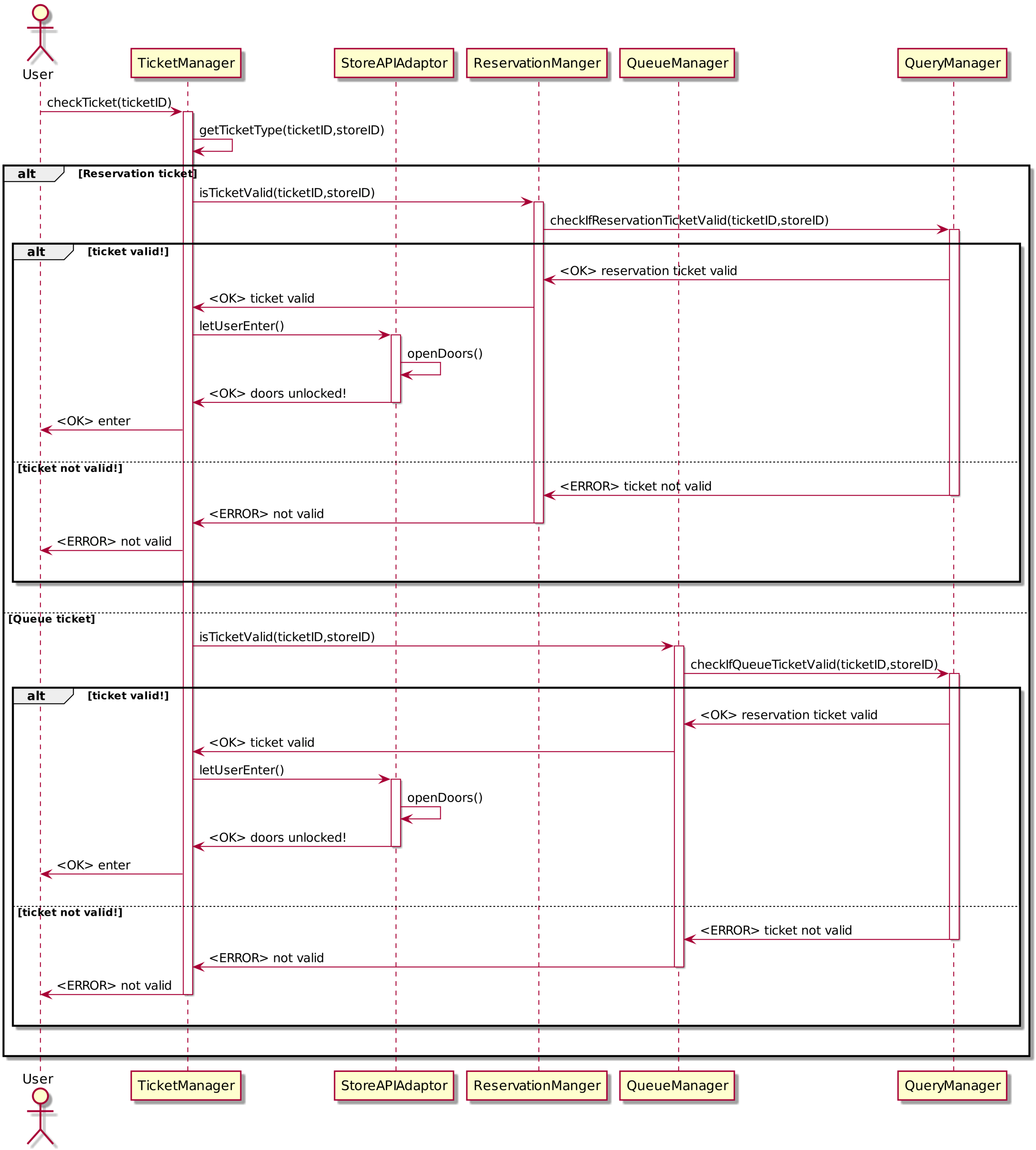

Figure 10: User access the store - Data Base is omitted for clarity

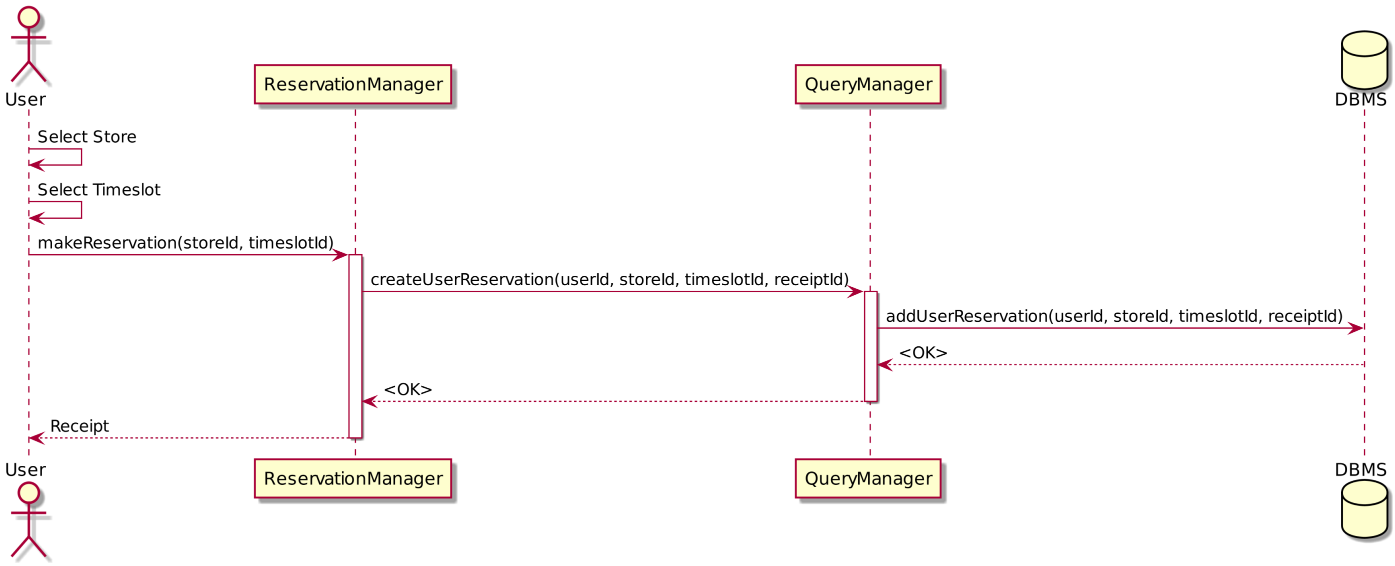

Figure 11: User makes a reservation

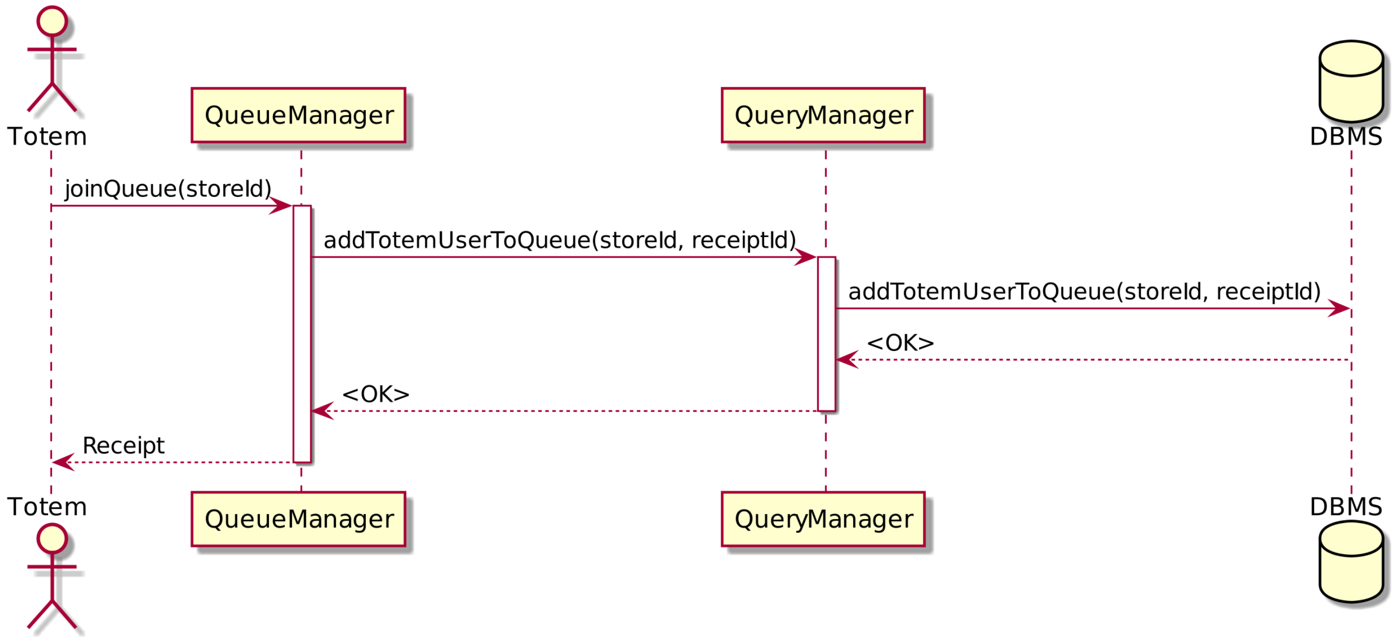

Figure 12: User joins the queue from the totem

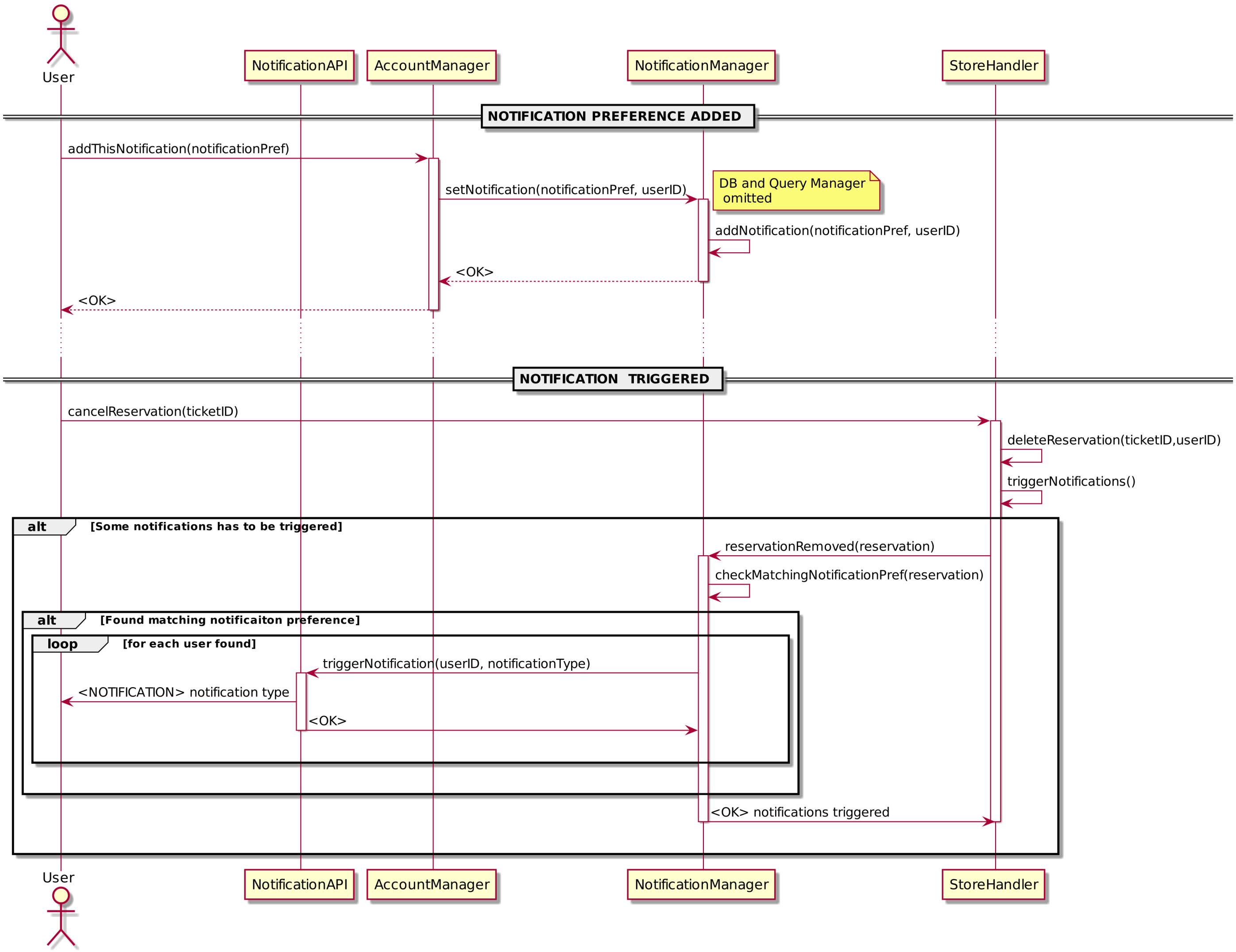

Figure 13: User sets a notification - User is notified

2.5 Component Interfaces

2.6 Selected Architectural Styles and Patterns

2.6.1 Architectural Styles

Thick Client The main characteristic of thick clients is offering a wide variety of functionalities independent from the central server. The main advantages it offers are greater decoupling of frontend and backend and a reduced computational effort on the application server. Recent years have seen a rise in the adoption of single-page applications, with the advent of cross-platform frameworks which allow to write code that can be run both in an app and in a browser. This allows developers to reuse significant portions of the codebase across a large number of target platforms, reducing the effort of keeping updated different versions of the same product. The single-page web application will be served by a dedicated static webserver, which is isolated from the rest of the system.

REST API REST is an architectural style centered around the definition of a uniform and predefined set of stateless operations defined on top of the HTTP protocol. Its main advantages are simplicity, scalability and modifiability. It allows to have a single endpoint against which both the mobile application and the web application can make requests, therefore eliminating the need of having multiple interfaces, while making it easier to maintain.

Three layer architecture Separating presentation, business, and data layers offers great flexibility, maintainability and scalability. This combined with a thick client means that the only communication between the client and the server goes through a predefined API, without having to worry about eachother's internal representation.

2.6.2 Patterns

MVC The software will be based on the *Model-View-Controller architecture*, where the model resides on the server, and the view on the client. A portion of logic is handled by the client to alleviate load on the server, for example filtering map results. The business logic is manager by the server.

Facade All services will be exposed in a reduced minimal API, hiding the real complexity of the system and making available only high level operations to the client.

Adapter The Query Manager component implements the Adapter pattern, as it mediates between the business logic and the DBMS services, exposing only a restricted and higher level set of functionalities.

2.7 Other Design Decisions

Maps The store search screen in the client applications will show the stores on a map for easier navigation and to show a clear view of all possibilities to the user. The maps will be offered by an external service capable of recognizing the addresses of the stores.

SMS During the creation of an account the user will be asked to insert their mobile phone number into the system. Then, the user will receive a confirmation code through an SMS sent via an external service, in order to certify their identity. This step is needed in order to mitigate the problem of the creation of fake accounts and fake reservations, which could be used by hacker in order to clog up the system and damage both stores and users.

3 User Interface Design

3.1 Client Interface

The *RASD* contains already several mockups of the mobile applications. In Fig. 14 we provide the flow diagrams describing the user experience of the client applications, which will be implemented by both the Mobile App and the Web App.

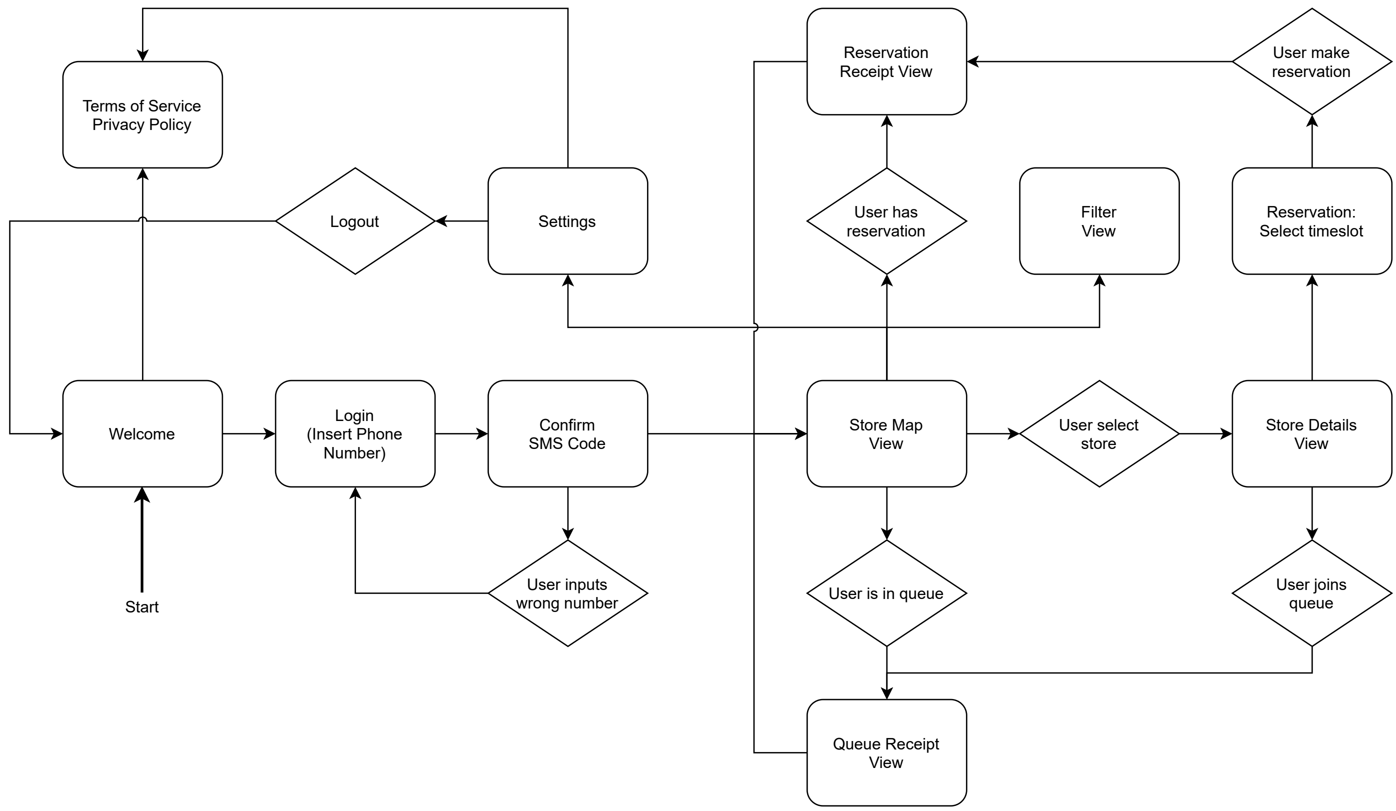

Figure 14: UX flowchart of application views

Additional details can be found in *Section 3.1.1* of the *RASD*. In the flowchart there are additional views such as a setting page or a viewer for the *Terms of Service* document that don't really need a mockup.

Users need to login in order to use the application, and the can logout through the settings page.

The web application interface will be mostly identical to the mobile application one, in order to provide a coherent experience.

3.2 Manager Interface

The admin web panel allows store managers to add, edit and remove stores. They need to login beforehand with credentials generated at deployment time. Managers can edit store details, timeslots, number of allowed customers, and view statistics about stores and users. The hierarchy of the views is illustrated in Fig. 15

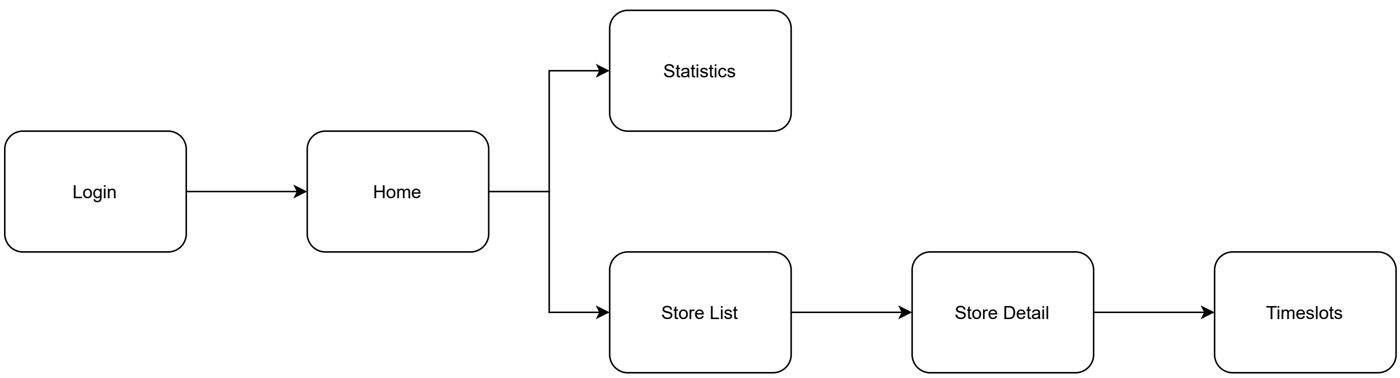

Figure 15: UX hierarchy of manager web panel

4 Requirements Traceability

In this section the requirements specified in the RASD are mapped to the components defined in this document. We'll only consider top-level components since they've already been explained. Frontend components are omitted for clarity.

Req.	Description	Components
R1	Allow a User to sign up for an Account after providing a mobile phone number.	**Account Manager**
R2	Allow a Registered User to find Stores nearby a specified location.	**Store Search**, **Maps API**
R3	Allow a Registered User to filter out stores based on available timeframes, days and distance.	**Store Search**
R4	Allow a Registered User to get in the virtual line at a specified store.	**Ticket Manager**, **Queue Manager**
R5	Allow a Totem User to get in the virtual line of the store where the totem is installed.	**Ticket Manager**, **Queue Manager**
R6	Allow a Registered User to preview an estimate of the queue time.	**Ticket Manager**, **Queue Manager**
R7	Allow a Registered User to book one visit to a specific store.	**Ticket Manager**, **Reservation Manager**
R8	Allow a Registered User to cancel their reservation.	**Ticket Manager**, **Reservation Manager**
R9	Allow a Registered User to leave the virtual queue.	**Ticket Manager**,**Queue Manager**
R10	Allow a Registered User and a Totem User to retrieve a scannable QR Code/Barcode that they must present in order to be granted access to a store.	**Ticket Manager**, **Queue Manager**, **Reservation Manager**
R11	The System notifies the Users affected by delay.	**Queue Manager**, **Reservation Manager**, **Notification API**
R12	The System cancels User reservations in case of a major delay.	**Queue Manager**, **Reservation Manager**, **Notification API**
R13	The System enforces the limits on the allowed number of concurrent Customers inside a store by restricting the access at the entry points (for example, automatic doors or turnstile).	**Ticket Manager**
R14	The System grants a User with a reservation access only within a short time (set by the manager) after the User's time of reservation.	**Ticket Manager**
R15	Allow System Managers to set the division of the maximum number of people allowed between the normal queue, the priority queue for people with special needs and the book a visit slot capacity.	**Admin Services**, **Reservation Manager**, **Queue Manager**
R16	The System calculates the average shopping time by recording every time a user enters and exits the store.	**Store API Adaptor**, **Ticket Manager**
R17	Allow System Managers to set a limit to the people allowed into the store at a time.	**Admin Services**
R18	Allow System Managers to choose the frequency and size of the time slots.	**Admin Services**, **Reservation Manager**
R19	Allow System Managers to know the average time spent in the store.	**Admin Services**
R20	Allow System Managers to know the current and past number of people in the store.	**Admin Services**
R21	Allow System Managers to check the current status of the queue and of the time slots.	**Admin Services**, **Reservation Manager**, **Queue Manager**

Table 1: Your caption here

5 Implemention, Integration and Test Plan

5.1 Overview

The application is composed of three decoupled layers (*client*, *business*, and *data*) which can be developed and unit tested independendly, and integration tested at the end.

Front-end components They consist of the mobile and web application, that have been presented in Chapter 2. They consist mostly of presentational components that belong to the client layer. Since both applications rely on a REST API, and should likely reuse portions of the codebase, they can be easily unit tested by mocking of the REST API.

Back-end components They are components that resides in the server, from both the business and the data layer.

External components They consist of the *Maps API*, the *SMS API*, and the *Notification API*. Since they're provided by third parties, they're supposed to be reliable and conform to their specifications.

5.2 Feature identification

To better plan the testing each component will require, it's useful to visualize them in a table (Table 2) where each components is associated with its difficulty of implementation and its importance for the system.

Feature	Importance	Difficulty
User login	High	Medium
Join queue	High	Medium
Reserve timeslot	High	High
Search store	Medium	Medium
View store details	Medium	Low
Notify customers	High	Medium
Adjust store parameters (managers)	High	Medium
Add/Remove/Edit stores (managers)	Medium	Low
View statistics (managers)	Low	Low

Table 2: Importance and difficulty of required features

5.3 Approach

All components will be implemented and tested with a *bottom-up* approach, in order to reduce the overhead that would have derived from a *top-down* one.

Components from the same subsystem (for example, the backend) can be implemented, unit-tested and integration-tested without a real need of components from another subsystem. This allows developers to develop in parallel the client and the backend, thus speeding up the development process.

Finally, after the final integration testing is complete, it's important to verify the adherence to the specified requirements.

In particular, the web and mobile application can be developed without need for a server, as the REST API can be easily mocked in tests.

5.4 Components integration

Here components and subsystems are illustrated via graphs where the arrow $x \longrightarrow y$ means "x depends on y". Subsystems are a group of components meant to be integration tested together after unit testing.

The first components to be tested together are the *Query Manager* and the *DBMS*, because it serves as the foundation upon all the other components rely when handling data.

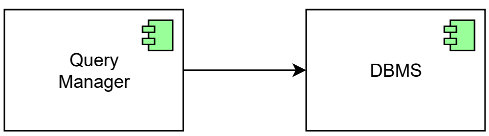

Figure 16: Data subsystem

The following three subsystems might be developed in any order, or simultaneously, as they do not depend on each other.
It is recommended to develop the *Store Handler Subsystem* first, as it represent the core functionality of the product. This is the item that should be the most carefully tested. It also requires the *Notification Manager* component to be implemented during integration testing.

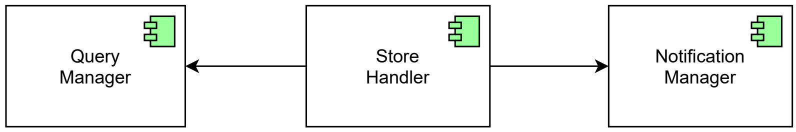

Figure 17: Store handler subsystem

The *Account Subsystem* is critical because it's needed to regulate the use of the product through the creation of the account. It should be tested with particular attention to user's input, handling invalid data correctly.

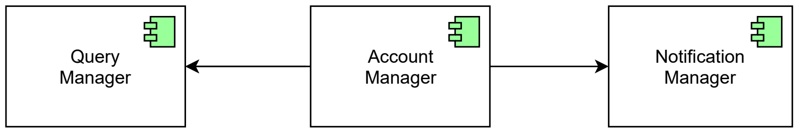

Figure 18: Account subsystem

The *Store Search Subsystem* is the least critical subsystem to be implemented, and it doesn't rely on any other component beside the *Query Manager*.

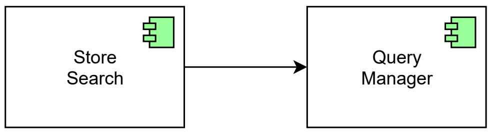

Figure 19: Store search subsystem

Once the critical subsystems have been implemented, it's the right time to implement and test the *Admin Services Subsystem*, which is the backend for all operations performable by store managers. It's important to test that all formal requirements are satisfied and cannot be violated by the user.

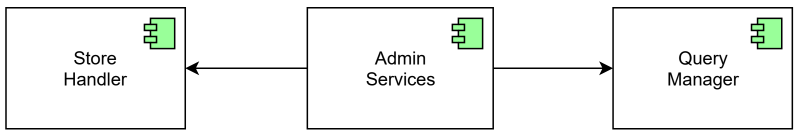

Figure 20: Admin services subsystem

At this point all subsystems have been implemented and tested, so the next step is performing integration tests between the client (front-end) and the server (back-end).
The client web and mobile app needs to interface with both the *Store Handler Subsystem* and the *Store Search Subsystem*.

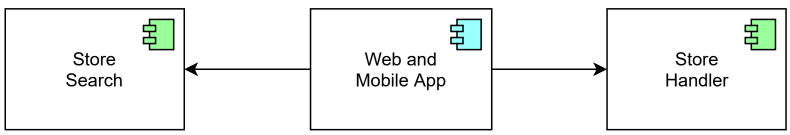

Figure 21: Client application integration

The totem needs to interface only with the *Store Handler Subsystem*, since the store it's predetermined at deployment.

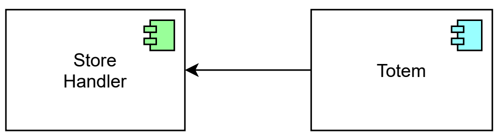

Figure 22: Totem integration

The *Admin Control Panel* has its own interface that should be tested.

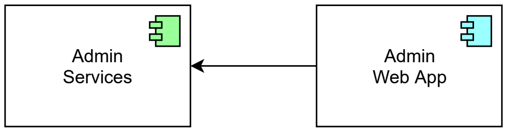

Figure 23: Admin panel integration

The final result of integration of front-end, back-end and external components is the following:

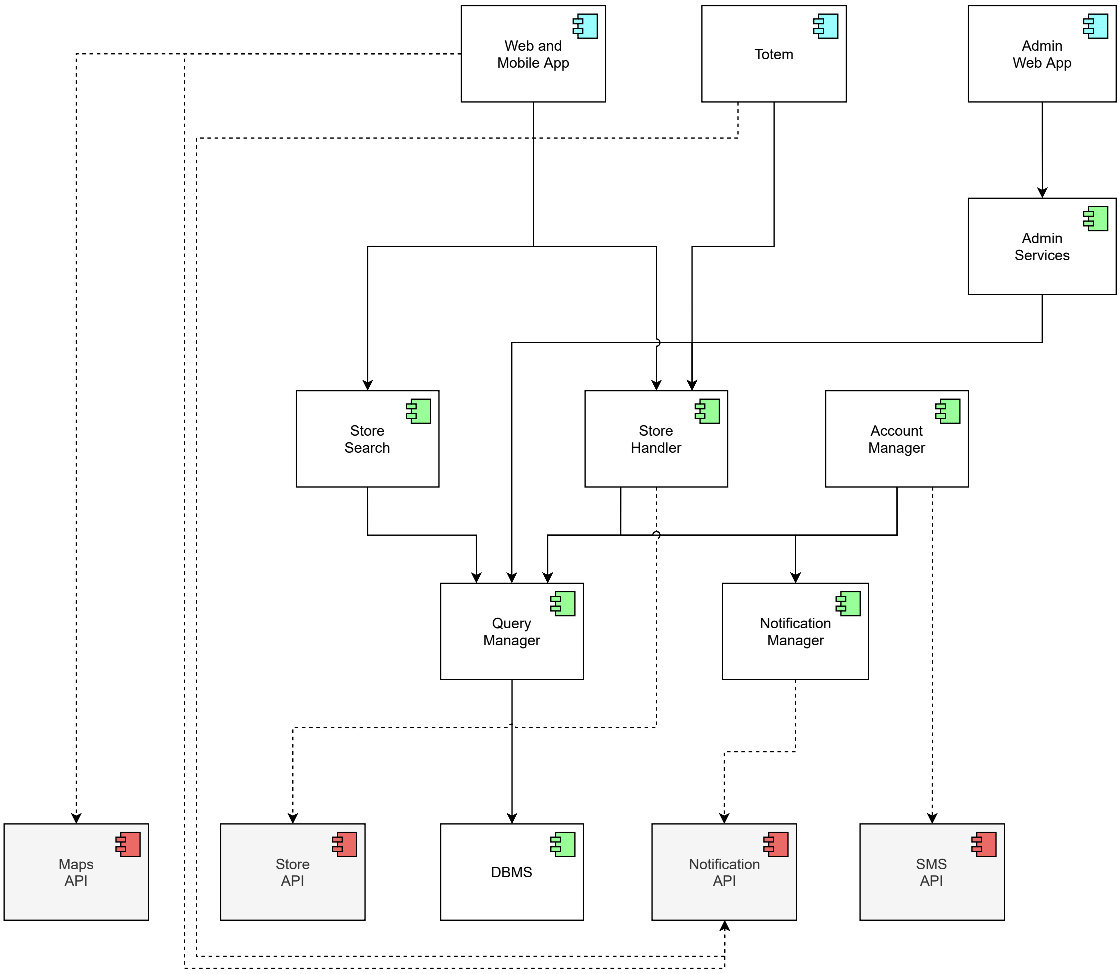

Figure 24: Full component and subsystems integration.
A dashed line means integration testing with external components/APIs.

6 Effort Spent

7 References