CSTS

Conceptual Architecture Report

Esra Kastrati - 215507205
Mostafa Mohamed - 215830292
Yassin Mohamed - 213867064
October 21st, 2020

Abstract

This report examines the conceptual architecture of the Common Spaces Tracking System. It starts off by examining the problem domain, the context of how the problem CSTS aims to solve came to be, and detailing the way it aims to solve it. The report details the users of the software system, and from that extracts all of the requirements. Those requirements are then used to decide on a software architecture for the system. We then examine different viable architecture patterns, seeing how compatible they are with our list of requirements, and select multiple architecture patterns to satisfy our software requirements. After examining different options, we establish that Client-Server is the best architecture to allow for concurrent and persistent access to the database. We then establish that we need two additional architectural styles: MVC and 3-tier architecture. Using this information, we draw out the use case diagrams for our main users, and the sequence and activity diagrams of the main use-cases.

Introduction

The world is undergoing a major shift in how people interact with each other. Social distancing guidelines and other similar measures meant to protect people have restricted the myriad ways we interact and socialize. Getting around these restrictions, and figuring out ways to let people interact the way they want is the purpose of many new software ventures, and the key to the success of many existing ones. While the inconvenience, inefficiency and strain of not being able to socialize are at the forefront of the perceived consequences of the Coronavirus pandemic, it has caused many more, harder to notice issues around the world. One such problem, and the one tackled by our software system, is the underutilization of pre-existing resources.

As physical distancing guidelines are imposed and strengthened, more and more buildings, campuses, libraries and other common spaces lie unused, a lot of people find themselves in positions where they have nowhere to go. Some are unable to work from home, due to family issues or inhospitable conditions, and some don't have the infrastructure that would allow them to accomplish their work from home. In both those cases, we have an abundance of resources around: the aforementioned libraries and campus buildings. However, we're unable to use them, as it entails the risk of infection with Coronavirus.

The issue in this case is simple, the government has not completely outlawed socializing in indoor environments, it has simply limited the number of people who can share one space. This means that the main barrier that stands between different locations, buildings, organizations and general study space from making use of their resources is an inability to effectively manage and keep track of the inflow and outflow of people into these spaces. Our software system offers a generic easily implementable solution to the underutilization of common spaces during the pandemic. Our system is only concerned with the software space, we offer clients a system that would allow them to present their common spaces to the public, and keep track of occupants by adding or removing them into the spaces; the specific method of controlling the inflow/outflow of people, through physical checkpoints or otherwise, is up to the client and outside our scope.

During the development lifecycle of any complex software system, the underlying architecture is discussed and different options are weighed carefully before one is chosen. This report discusses the abstract architecture of our software system, it details the main subsystems and components of the system and how they interact. Based on that description, it examines the different software architecture patterns that could be applied, contrasts between them, and outlines the chosen architecture of the system. Initially, we will give an overview of the system's users as well as the requirements.

Requirements

Seeing as our system is intended to allow organizations to make use of their available common spaces, we can outline three main users:

Admins for management of the system: Complete control over the database. Server shutdown, creation of databases, managing or transferring databases.
Client: Interact with database through client interface and utilities (database API).
User: These users examine the results that come from the database (queries).

Functional Requirements:

REQxF-1	high	Clients can login to the application
REQxF-2	high	Clients can change maximum capacity of given Common Space
REQxF-3	med	Clients can view a list of occupants for a common space
REQxF-4	high	Clients can add/remove users to a common space
REQxF-5	high	Clients can add time-limits to visiting Common Spaces
REQxF-6	high	Clients can ban users from accessing database
REQxF-7	med	Users can login to the application
REQxF-8	high	Users can browse available Common Spaces
REQxF-9	high	The System Administrator can add/remove Client access
REQxF-10	med	The System Administrator can create and shutdown databases

Non-functional requirements:

Performance:

REQxUF-1: the system will display common spaces within 2 seconds of request

Security:

REQxUF-2: Users cannot make changes to database.

REQXUF-3: Users cannot access other user’s information

Usability:

REQxUF-4: the system should be accessible through web or mobile application

Scalability:

REQxUF-5: the system should withstand a minimum of 20 concurrent users.

Availability:

REQxUF-6: The system must run 24x7x365, with overall availability of 0.99.

Reliability:

REQxUF-7: Data returned should always be consistent with database.

Architecture

From a technical standpoint, our system manages a database containing records of available common spaces, as well as up-to-date information regarding the current occupants of any common space. It aims to offer clients an easy solution to integrating this management system with their physical resources. This means that for our system, we are going to have multiple entities seeking to access and edit the same pool of information. In addition, we have a different type of entity that can only access the information but not edit anything. For our purposes it makes sense to have this shared pool of information as a component, specifically a Database Server. This will allow it to be accessed by multiple users concurrently. It also follows that it must be remotely accessible, both by managers performing admin tasks and users looking to query the state of the common spaces, thus the database will be connected to other entities through a network. Following this, the other main component of the system is the Clients, which will need to concurrently access the Database.

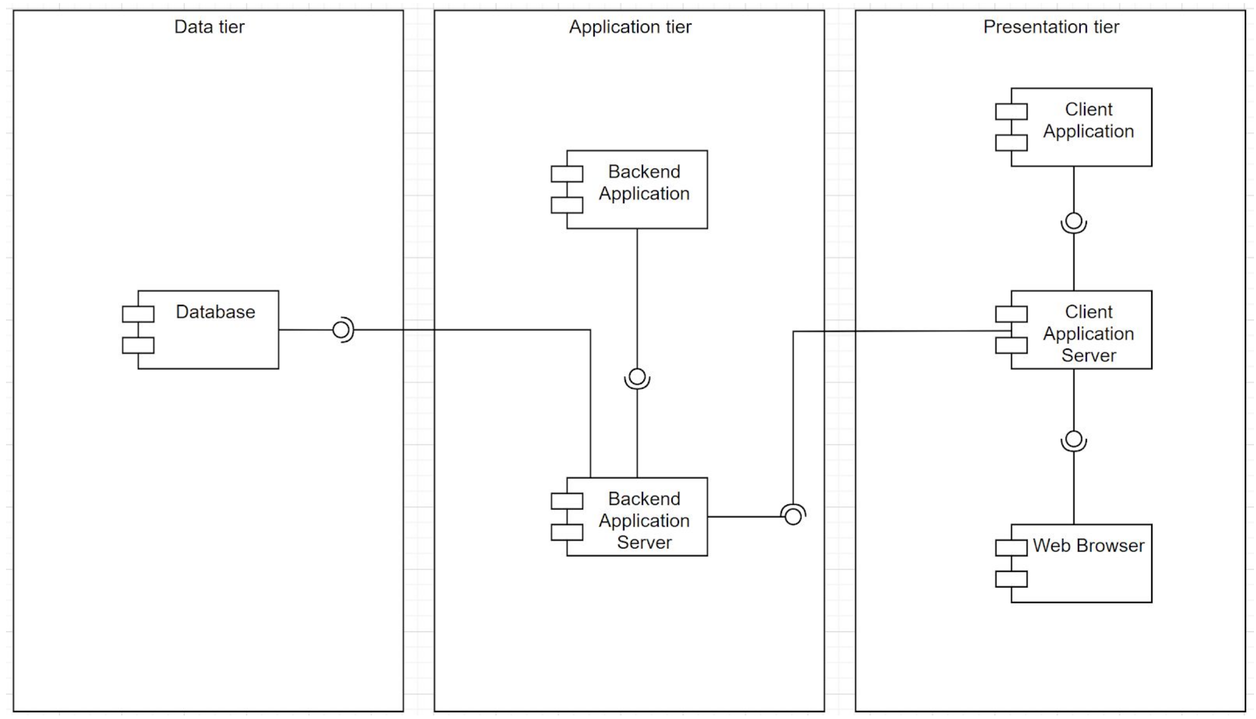
From these components we determined that the top-level architecture we will use for Database communication is the Client-Server architecture. With our main components being the clients accessing the Database, and the database itself. The alternative approaches we could take in this case are to have a local database where clients would access the database through physical means or local networks. However this would not allow users to access the database remotely, and so is not feasible. Alternatively, a decentralized database would jeopardize the security and privacy of the information.

Client-Server is not the only architecture pattern present, it only covers communication between our client end-points and the central database. Within the client component, a User Interface is running that would allow the managers to add or remove occupants from the common spaces, add or remove common spaces, and check who is currently occupying those same spaces. In addition, the UI can display the information multiple ways, and so the application can perform some operations on the data after it was retrieved from the database. For this situation, we examined multiple different architecture styles for building the application’s back-end, front-end and connecting them together and to the database. Initially, we knew we had to apply an architecture pattern, as our application was too complex to be done with no overarching structure. Since our application has a User Interface, we initially thought of implicit invocation. However, we realized that the implicit invocation pattern does not lend itself to UI applications and, more importantly, it’s only useful in disseminating information one way, not both ways as is necessary in this case. Finally we examined the layered application structure. While it seemed a good fit for our application in terms of separation of concerns, we found that it was too restrictive in terms of communication between our different software components. Which led us to pick MVC for the internal architecture of our system. MVC allows us to separate the business logic from the view, and be able to perform complex operations on the data and have complete freedom over how it is displayed. In addition, this separation allows us to dedicate a specific access point inside the application to communicate with the database.

At this point we had two main architectural styles we knew would be implemented in our system: Client-Server for communication with the Database, and MVC for the application itself. However, while these two architectural styles fit our needs perfectly, the overall structure of the system seemed disjointed; we had no distinct structure that encompassed both the database and the application. We realized that our application was the perfect candidate for a 3-tier architecture. Our application contained the view, which is the presentation tier, the model and the controller, which are the business tier, and the application is connected to the database, which is our data tier.

Following that, we established our overarching software architecture. The application overall is split into 3-tiers through the 3-tiered software architecture, the Presentation and Application tiers both represent one instance of our clients, and are connected to the Database through the Client-Server architecture, allowing clients to query the database remotely; within our Presentation and Application layers, we use the Model-View-Controller architecture to separate view from the logic, and allow us to build a more robust UI.

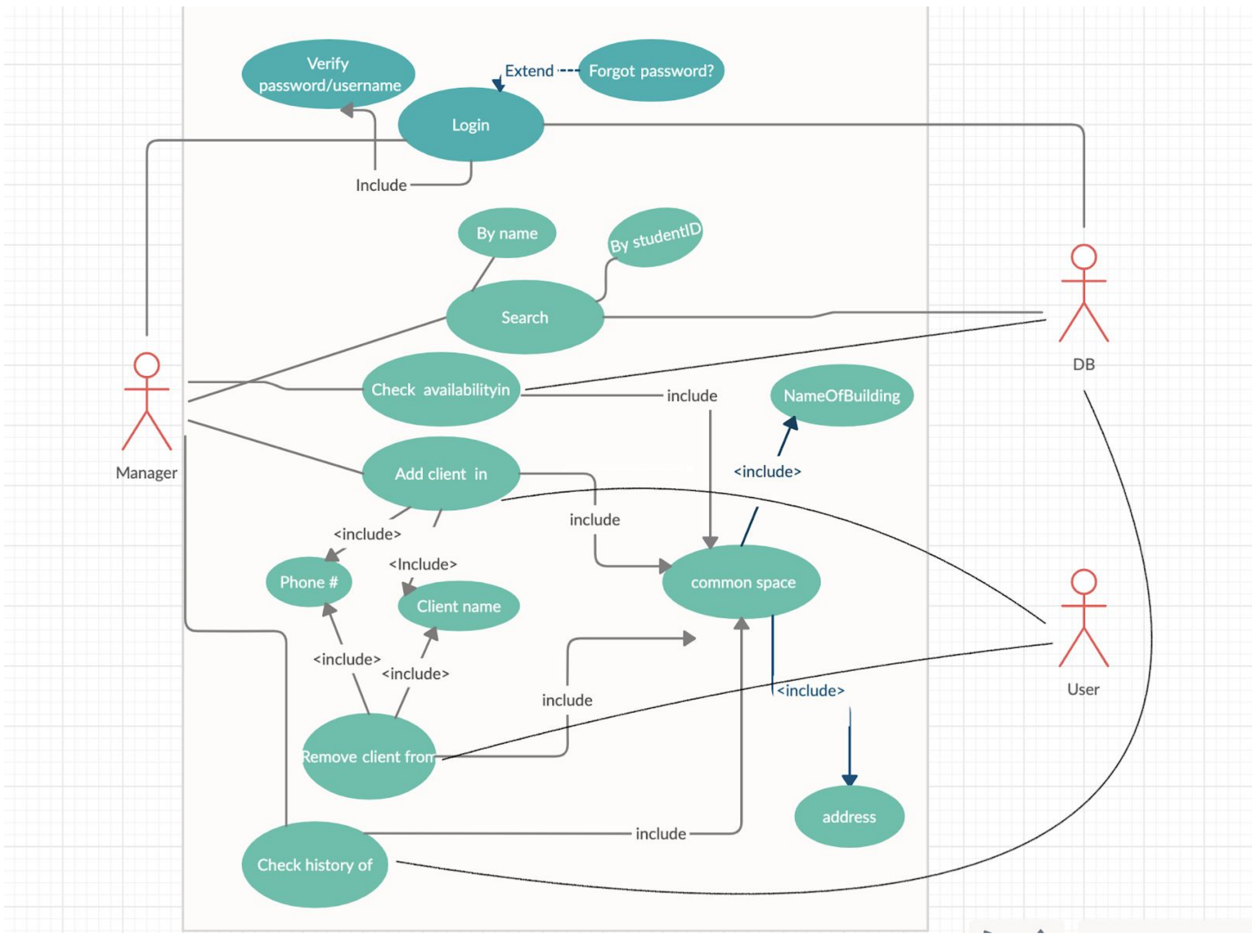
Below is a component diagram showing the high-level architecture of our system:



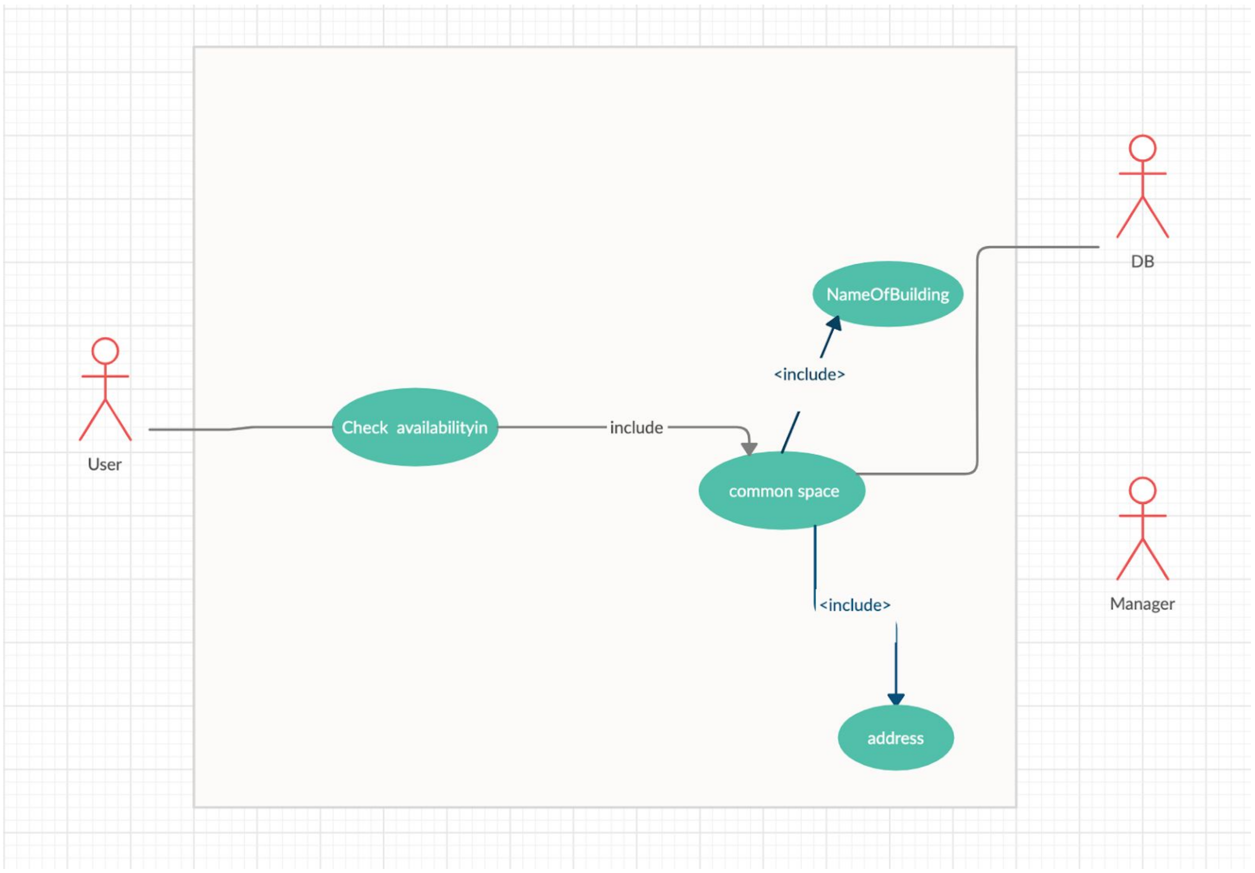
Due to the specific nature of our project, we could not find anything that tackles the same problem. In terms of technical implementation, however, there are many projects that make use of the design patterns we did, mainly anything with a central database that needs to be accessed by multiple users concurrently. Upon further research, we find that most of these fall under the architecture of “Active Database”, which is an architectural style that encompasses many of the ones we used in our system.

Use cases

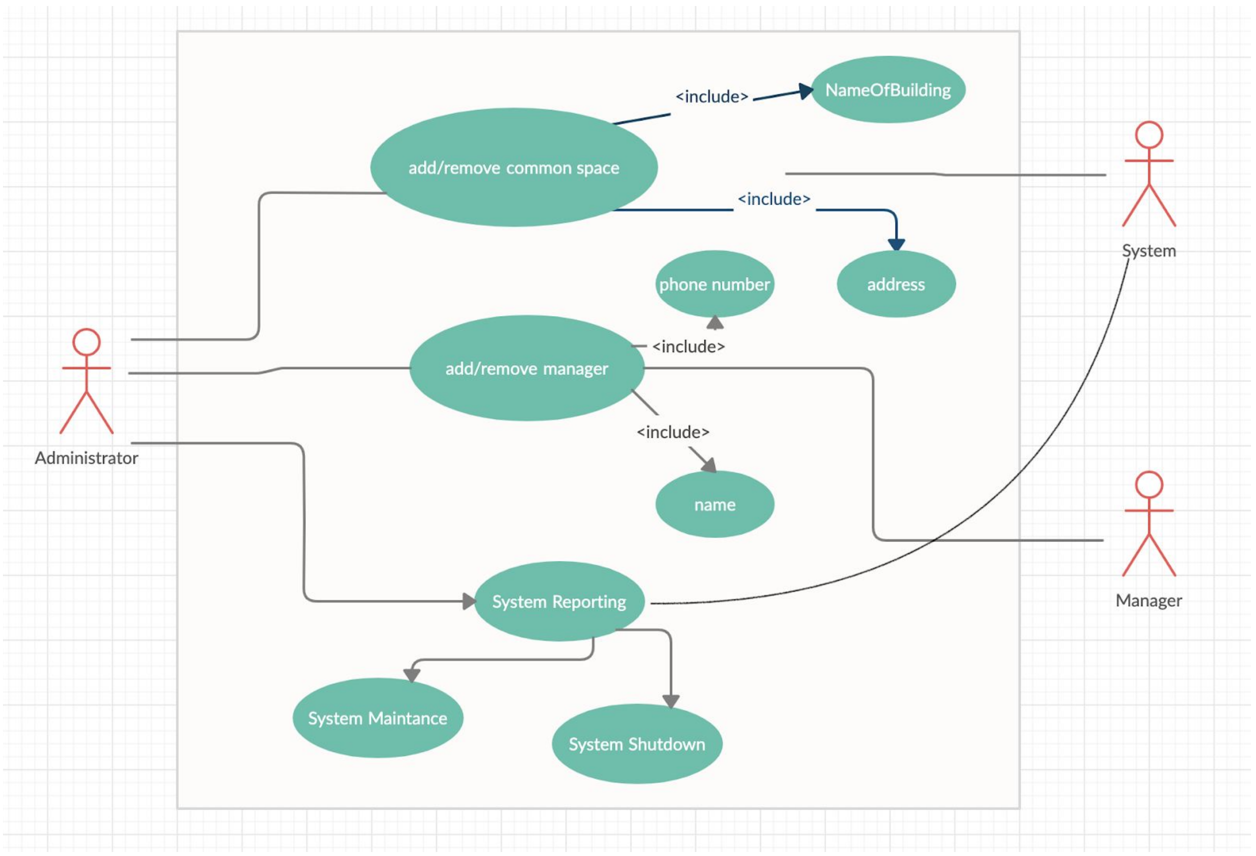
The manager use case diagram below illustrates the importance of remote access to the Database, as they can perform the most operations on the data, while also hiding the implementation details from the Users who just need to query the Database.



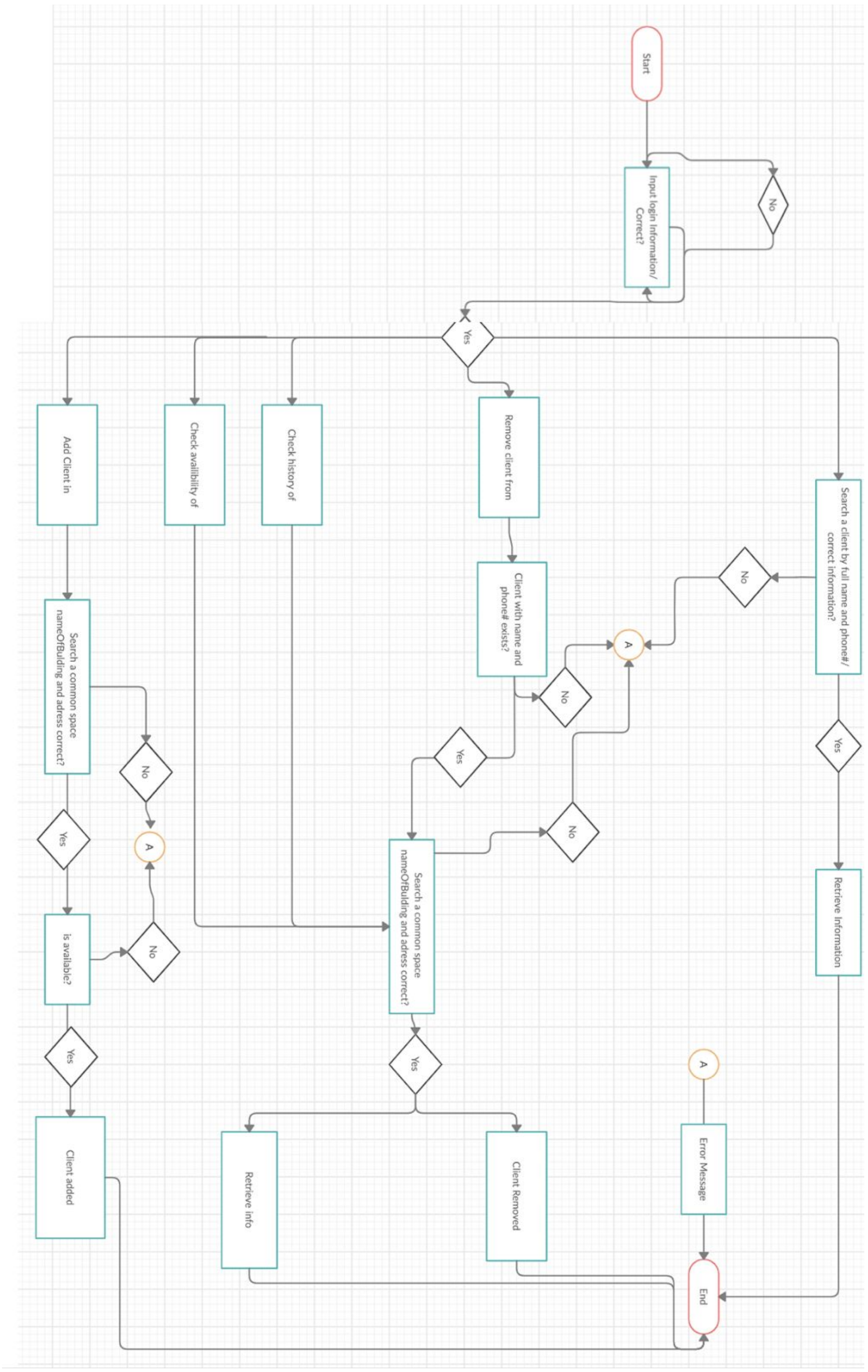
Use-case diagram for a Manager (Client) of the system.



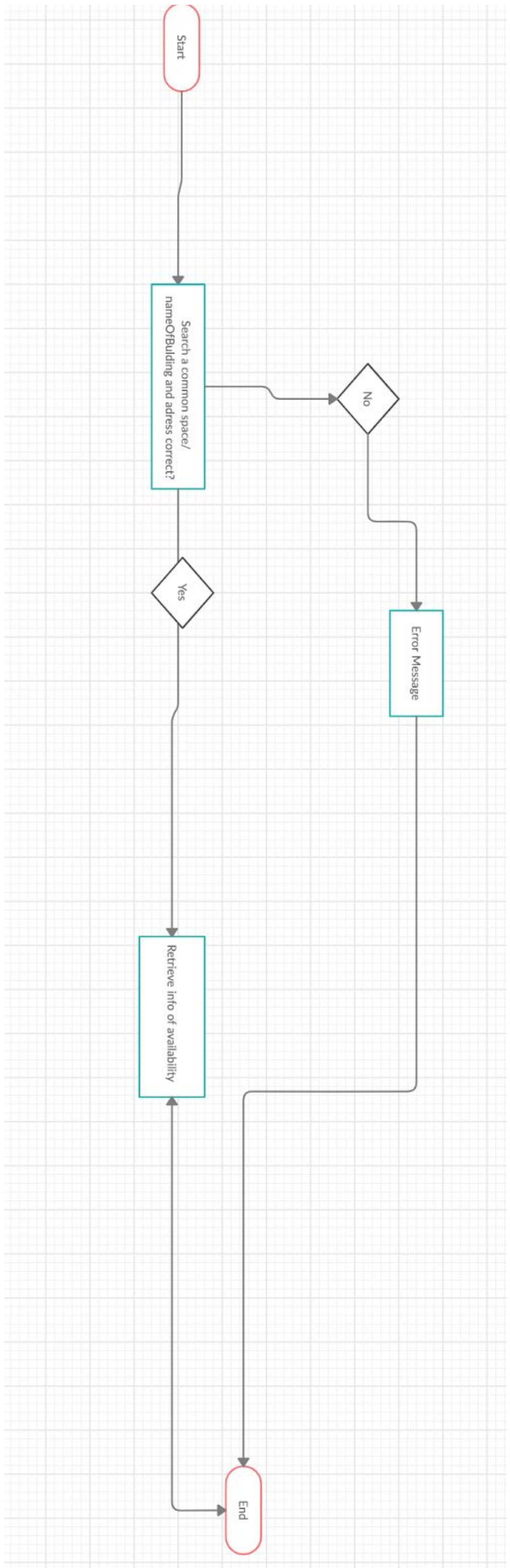
Use-case diagram for a User of the system.



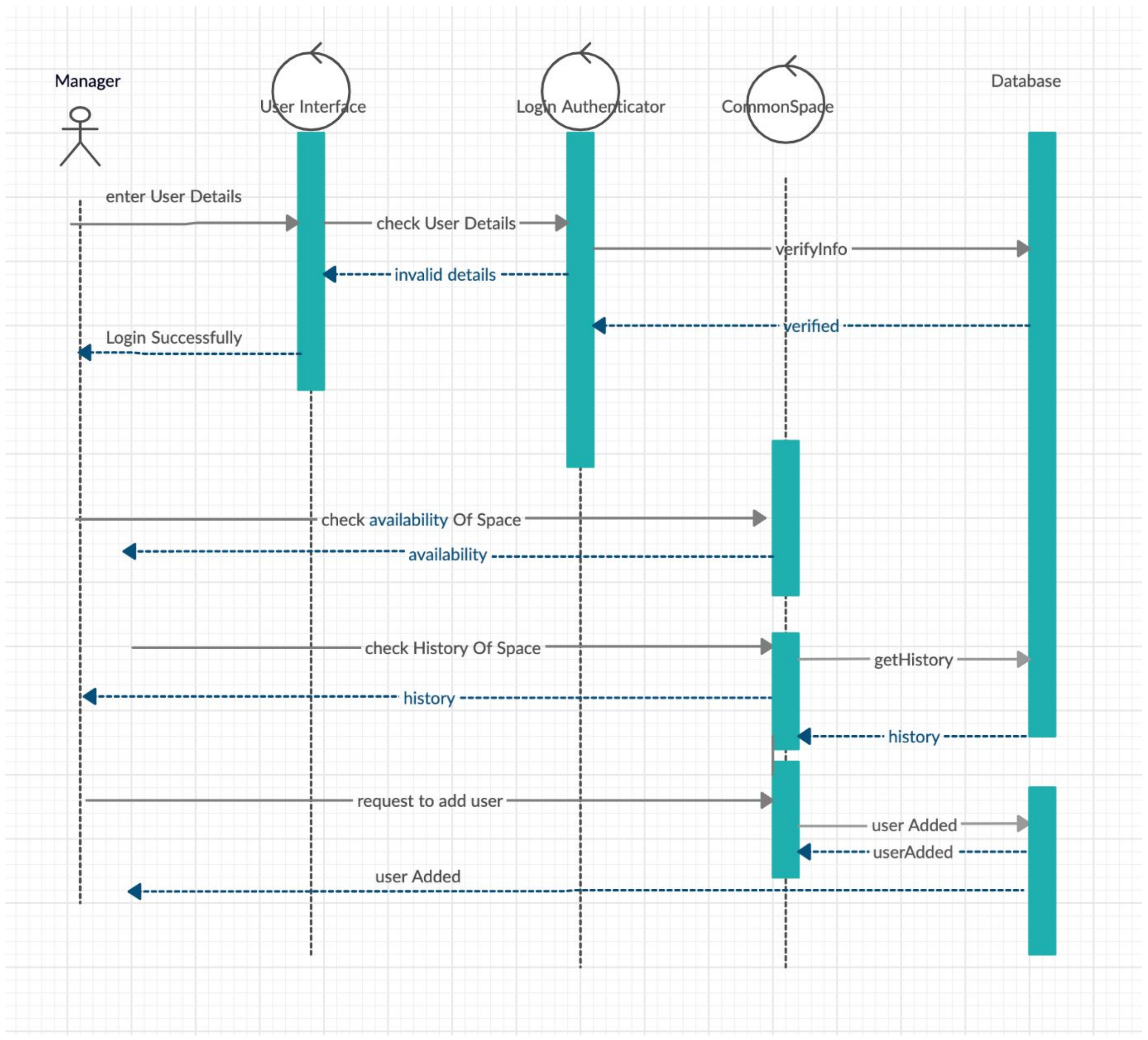
Use-case diagram for an Administrator of the system.


Start
Input login Information/ Correct?
No
Yes
Add Client in
Check availibility of
Check history of
Remove client from
Search a client by full name and phone#/ correct information?
No
A
Client with name and phone# exists?
No
Yes
Yes
Search a common space nameOfBulding and adress correct?
No
A
Yes
is available?
No
Yes
Client added
No
Search a common space nameOfBuilding and adress correct?
Yes
Retrieve info
Client Removed
A
Error Message
End
Retrieve Information

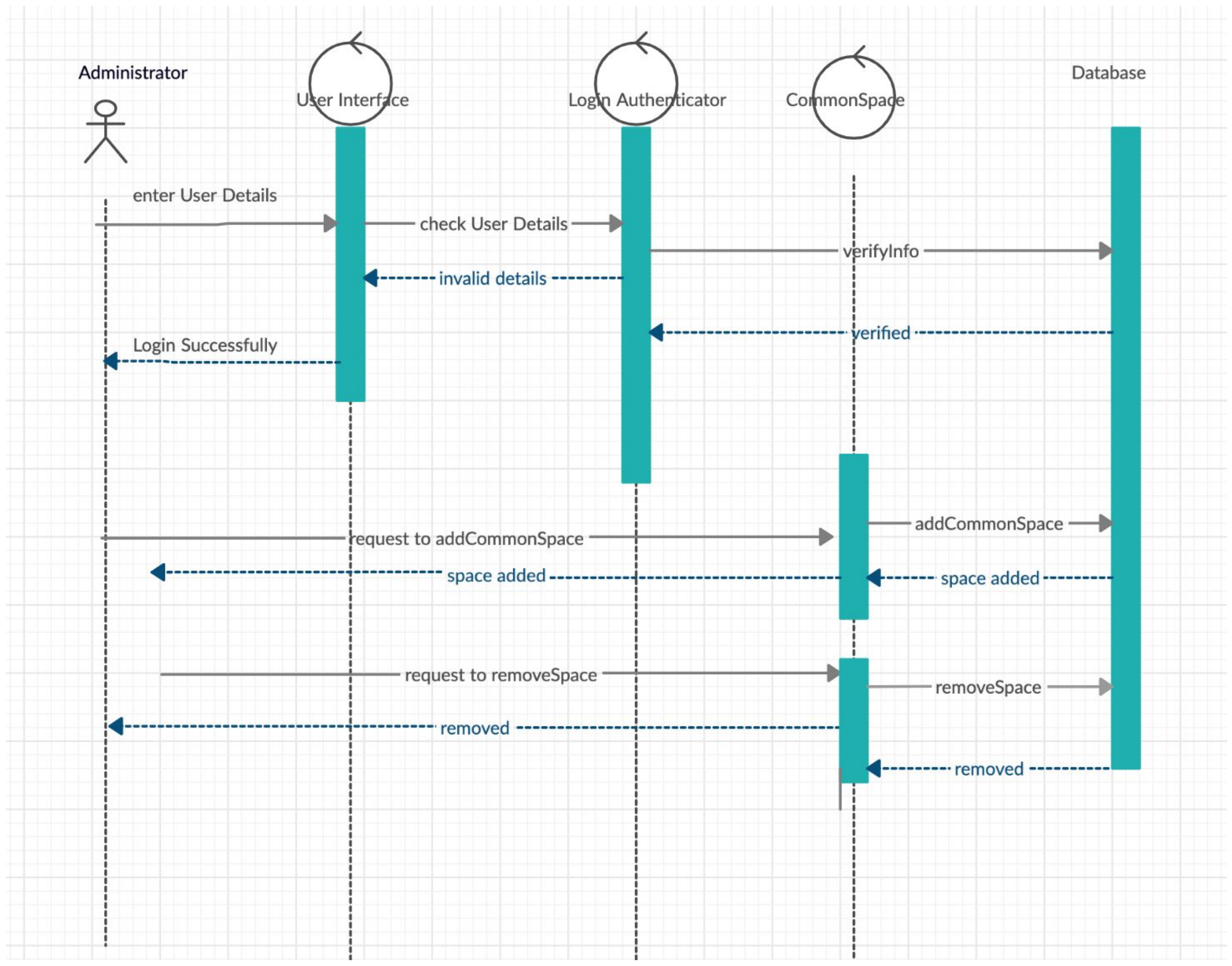
Activity diagram for a Manager (Client) of the system.


Start
Search a common space/ nameOfBulding and adress correct?
No
Yes
Error Message
Retrieve info of availability
End

Activity diagram for a User of the system.



Sequence diagram for a Manager of the system.



Sequence diagram for an Administrator of the system.

Conclusion

As software systems get larger and larger, the need for established and tested software architectures increases as well. While we were conceptualizing our software, we had to think again and again about what architecture styles are available to us, and compare and contrast between them. Initially, we determined the users of our software, and from that elicited all the requirements. That allowed us to start examining the needs of the users and what software architectures would be best suited to fill them. Since the paramount function of the system is to be available, we decided on a client-server architecture, so that different clients can access and query the database at the same time. We also implemented MVC architecture, which is necessary for creating complex web applications and interactive UI. Finally, we applied the 3-tier software architecture to our overall system, ensuring that there’s no redundancy and maintaining separation of concerns.

This development activity really outlined the importance of following the design method, for instance defining the users at the beginning is what allowed us to come up with comprehensive requirements, in turn, those requirements were critical for determining what software architectures would be viable in our system. In addition, this activity showcased why any complex software system would need to implement multiple software architectures in order to function efficiently and effectively.