POLITECNICO
MILANO 1863

Software Engineering 2
CLup project by
Robert Medvedec
Toma Sikora

Design Document

Deliverable:	DD
Title:	Design Document
Authors:	Robet Medvedec, Toma Sikora
Version:	1.3
Date:	9-January-2021
Download page:	https://github.com/robertodavinci/Software_Engineering_2_Project_Medvedec_Sikora
Copyright:	Copyright © 2021, R. Medvedec, T. Sikora – All rights reserved

Contents

Table of Contents **3**

List of Figures **4**

List of Tables **4**

1 Introduction **5**
1.1 Purpose 5
1.2 Scope 6
1.3 Definitions, Acronyms, Abbreviations 7
1.3.1 Definitions 7
1.3.2 Acronyms 7
1.3.3 Abbreviations 7
1.4 Revision History 8
1.5 Reference Documents 9
1.6 Document Structure 10

2 Architectural Design **11**
2.1 Overview 11
2.2 Component view 13
2.3 Deployment view 17
2.4 Runtime view 19
2.4.1 Book a visit 19
2.4.2 Retrieve a number 21
2.4.3 Store manager login 22
2.5 Component interfaces 24
2.6 Selected architectural styles and patterns 27
2.7 Other design decisions 29

3 User Interface Design **30**
3.1 Overview 30
3.2 Proposed design 31

4 Requirements Traceability **35**

5 Implementation, Integration and Test Plan **38**
5.1 Overview 38
5.1.1 Importance of features 38
5.2 Implementation 39
5.3 Integration strategy 40
5.4 Testing 43

6 Effort Spent **44**

List of Figures

1 Detailed three-layer system architecture . . . 12
2 Main component diagram . . . 14
3 Deployment diagram - System architecture . . . 17
4 Sequence diagram 1 – Book a visit . . . 19
5 Sequence diagram 2 – Retrieve a number . . . 21
6 Sequence diagram 3 – Store manager login . . . 22
7 Interface diagram . . . 26
8 Three-layer system architecture . . . 27
9 Model-view-controller . . . 28
10 Android icon concept . . . 31
11 iPhone icon concept . . . 31
12 Android home screen . . . 31
13 iPhone home screen . . . 31
14 App main screen . . . 32
15 App store selection . . . 32
16 App store manager login screen . . . 32
17 App ticket request screen . . . 32
18 App "Book a visit" screen . . . 33
19 App additional info screen (location on) . . . 34
20 App additional info screen (location off) . . . 34
21 App store manager screen (valid scan) . . . 34
22 App store manager screen (invalid scan) . . . 34
23 Integration diagram 1 . . . 40
24 Integration diagram 2 . . . 40
25 Integration diagram 3 . . . 41
26 Integration diagram 4 . . . 41
27 Integration diagram 5 . . . 42

List of Tables

1 Requirements table . . . 35
2 Components table - Goal 1.1 . . . 36
3 Components table - Goal 1.2 . . . 36
4 Components table - Goal 2 . . . 36
5 Components table - Goal 3 . . . 37
6 Components table - Goal 4 . . . 37
7 Components table - Goal 5 . . . 37
8 Importance of features . . . 38
9 Effort spent - Robert Medvedec . . . 44
10 Effort spent - Toma Sikora . . . 44

1 Introduction

1.1 Purpose

This document provides a detailed view of the architecture and the user interface design of the CLup system. Building on the RASD document, it gives a more refined technical and functional description of the system, explaining it on a much lower level.

To provide the full description of the system, UML diagrams will be used since they are the de facto industry standard. While the actual implementation is not part of the document, it outlines the presumed implementation, integration, and test plan, to help the software development team in the realization of the application.

The purpose of the document is primarily to guide software developers, but it can also provide useful information to end users and investors.

1.2 Scope

CLup is a simple application that helps store managers with handling large crowds inside their store and store customers with planning more efficient and safe grocery shops. The target audience for this application includes every person that shops for groceries in a store, which includes almost all demographics fall into this category.

Faced with a worldwide pandemic of the COVID-19 virus countries across the world imposed strict health measures in line with the recommendations of the WHO. To combat the spread of the virus, governments introduced decrees that limited the movement of the population to a certain degree. Only essential movement, such as: going to work, grocery shopping or outdoor exercise, was deemed acceptable. Although successful in the mitigation of the disease, the act put a serious strain on society on many levels. To help reduce the stress and anxiety, many aspects of everyday life involving close contact can be considered and improved upon.

This project aims to help with, and resolve the issues surrounding grocery shopping. As we all know, grocery shopping is an essential activity which involves close contact inside the store. Since the COVID-19 virus spreads mainly through airborne particles, this activity plays a key role in its mitigation. To reduce crowding inside the stores, supermarkets need to restrict access to their store and keep the number of people inside below the optimal maximum capacity.

The main idea is to enable store customers to enter a queue from home (or wherever they find themselves) through simple interaction with the application. Besides that, the application will give customers the option to "Book a visit" to the grocery store. This feature will allow them to view available time slots for their grocery shop, book the most convenient one, and optionally indicate an approximated duration of their visit to further improve the accuracy of the waiting time estimation of the system.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- **Application**: a computer (mobile) program that is designed for a particular purpose.
- **QR code**: a machine-readable code consisting of an array of black and white squares, typically used for storing URLs or other information for reading by the camera or a scanner.
- **Smartphone**: a mobile phone that performs many of the functions of a computer, typically having a touchscreen interface, internet access, and an operating system capable of running downloaded apps.
- **Google Maps**: a web mapping service developed by Google, used both as a standalone app and as an integrated mapping solution in most of the apps.
- **iOS**: operating system developed by Apple, used by their portable devices like iPads and iPhones.
- **Android**: most popular operating system for smartphones and tablets, developed by Google and partners.

1.3.2 Acronyms

- **RASD**: Requirement Analysis and Specification Document
- **COVID-19**: Virus responsible for the spread of the coronavirus disease 2019
- **CLup**: Customer Line-up
- **API**: Application programming interface, computing interface which defines interactions between multiple software intermediaries
- **WHO**: World Health Organization
- **GUI**: Graphical user interface
- **DB**: Database
- **REST**: Representational state transfer - software architectural style used in web services
- **DAO**: Data access object
- **JDBC**: Java Database Connectivity, API used in Java programming language

1.3.3 Abbreviations

- **Gn**: nth goal.
- **Rn**: nth functional requirement.
- **App**: Application.

1.4 Revision History

- **Version 1.0**: First .tex document created and added all together; 5th January 2021
- **Version 1.1**: Added interface diagram and fixed some errors, minor tweaks; 6th January 2021
- **Version 1.2**: Fixed grammatical errors, added effort spent tables, minor tweaks; 7th January 2021
- **Version 1.3**: Minor tweaks; 9th January 2021

1.5 Reference Documents

- Specification document "R&DD Assignment A.Y. 2020-2021.pdf"
- Presentations Software Engineering 2, Politecnico di Milano
- Star UML - Program used for creating diagrams
- Fundementals of Software Engineering - C. Ghezzi, M. Jazayeri, D. Mandrioli

1.6 Document Structure

This document is divided into six different chapters, each one further specifying and clarifying the system proposed in the RASD document.

The first chapter starts with the explanation of the purpose of this document and a brief recap of the end product. It also contains formal necessities such as the definitions, acronyms, and abbreviations for better understanding of the matter, the document revision history, the list of referenced documents, and this subchapter, the document structure.

The second chapter provides a detailed and implementation ready architectural design of the application. Firstly, the overview of the high-level components and their interaction is given, followed by the layout and the definition of the specific components of the system. Furthermore, it explains the deployment and the runtime view, in order to explain the way the components interact and how they relate to the needed use cases. In the end, the communication between the systems components, selected architectural styles and patterns and some other design choices are also explained in detail.

The third chapter revolves around the design of the user interface. To get a sense of what the finished product should look like, this chapter provides an overview on how the user interacts with the system and how the system will look like. This matter was already introduced in the RASD document, and here, expanding that description, the final thoughts and design aspects are presented.

The fourth chapter lists the requirement traceability by mapping requirements, goals, and components. It ties the core ideas of the RASD and the DD document together, by coupling the goals and the requirements listed in the RASD, with specific implementation components defined in the DD.

The fifth chapter contains our thoughts and guidelines for the actual implementation, integration, and testing of the system. It can serve as a development plan for the software engineers creating an application according to these documents. The chapter identifies the order in which the subcomponents are to be implemented, and at which points in time each one of them should be integrated. In the end, it provides a thorough overview of the test plan of the system as a whole.

The sixth part provides information about the number of hours each group member has spent working on each part of this document.

2 Architectural Design

2.1 Overview

Architectural design of the application is based on the widespread three-layer model that is used in most applications. Those three layers are presentation layer, also known as frontend, application layer, also known as middleware, and data layer, also known as backend. Every one of those three layers does its own part of the job and communicates with other two layers, directly or indirectly, to transfer necessary data and present needed information to the user.

This software design pattern is also very similar to MVC (Model-view-controller), which is gathering information and presenting it to the user differently based on their needs.

This architecture is a typical client-server implementation where server holds the data, and the client is accessing it through requests.

There are two main approaches to this design – thin client and thick (fat) client. Since one part of the work is done on the server and other part on the client's device, it is important to determine which part is going to do more work. We have decided to go with the thin client design, in which almost all the work is done on the server, including database management and additional computations, which are then presented to the client who only must host the application and be able to see the data. Since our application is not very complicated and does not require a lot of computation the servers do not have to be as powerful to do a lot of work. Also, our goal is that this application can be ran on pretty much every mobile device, therefore having a thin client ensures that mobile devices used to run it do not have to be very powerful. Lastly, since most of the work is required to be in sync with other data, it is better suited that all the computations are done at one place and then sent out to all the clients, rather than having clients do computations and then communicating to the server and back. That will greatly simplify syncing data and improve the speed of transferring data. However, it requires the Internet connection to be present at all times and that both client and server have constant communication in order for the application to work as desired.

In the following illustration, the concept is being shown along with the direction of communication between the major layers.

Presentation layer – Via application interface displays messages and system options to users, gathers input, and forwards it to the application layer. Minimal computation is done here due to thin client design. It receives the data from data layer via application layer and handles interaction with user.

Application layer – Transfers data between presentation and data layers, handles functions in the presentation layer and arranges data from data layer that is to be forwarded to the client. Communicates with Google Maps API to gather user's current location, computes necessary time variables, and sends it to the presentation layer.

Data layer – Stores and manages data within database, arranges received data and sends requested data. Communicates only with the application layer. Responsible for computing the data that is only in the database.

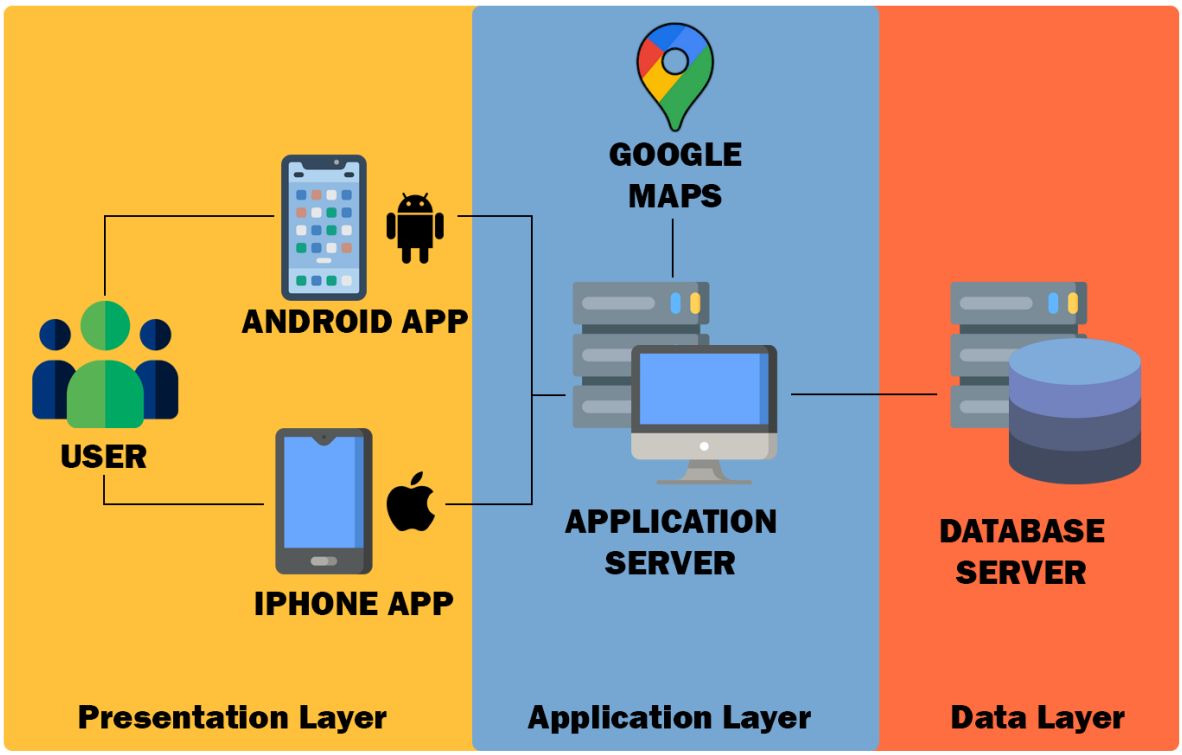

Figure 1: **Detailed three-layer system architecture**

Most of the additional variables, like estimated wait time and queueing are done within the data layer through the database. The data is calculated in real-time and stored directly into the database, so it is accessible to every client who requests it, with the application layer just being responsible for the data transfer. Since there are no user accounts in the application, all sent data is independent and there are no private variables for certain users (other than store managers of different stores). This improves security and privacy since no personal or vulnerable data is stored within the database. Nearly every piece of information is available to everyone, which improves speed and simplicity of the whole system due to not having to check user information by using ID tokens or some other technique.

2.2 Component view

In this section, a more thorough and detailed view of all of the main components is displayed. Since a lot of the work is being done in the application server part on the server, we mostly focused on that part to give a precise sketch of how some smaller parts of the system are connected and in which directions they communicate. As you can see in the component diagram, it is a very complex structure containing multiple major components and interfaces, external service, and several sub-components that are operating under one of the major ones. MobileApp and Database components are simplified and are not shown in great detail.

Component list:

- DBService
- Director
- RequestManager
- LoginManager
- StoreSelectionManager
- StoreManager
- TicketService
- QueueService
- ScheduleService
- DistanceService
- BookAVisitService
- EnterService
- ExitService
- AndroidApp (external)
- IPhoneApp (external)
- DB (external)
- GoogleMapsService (external)

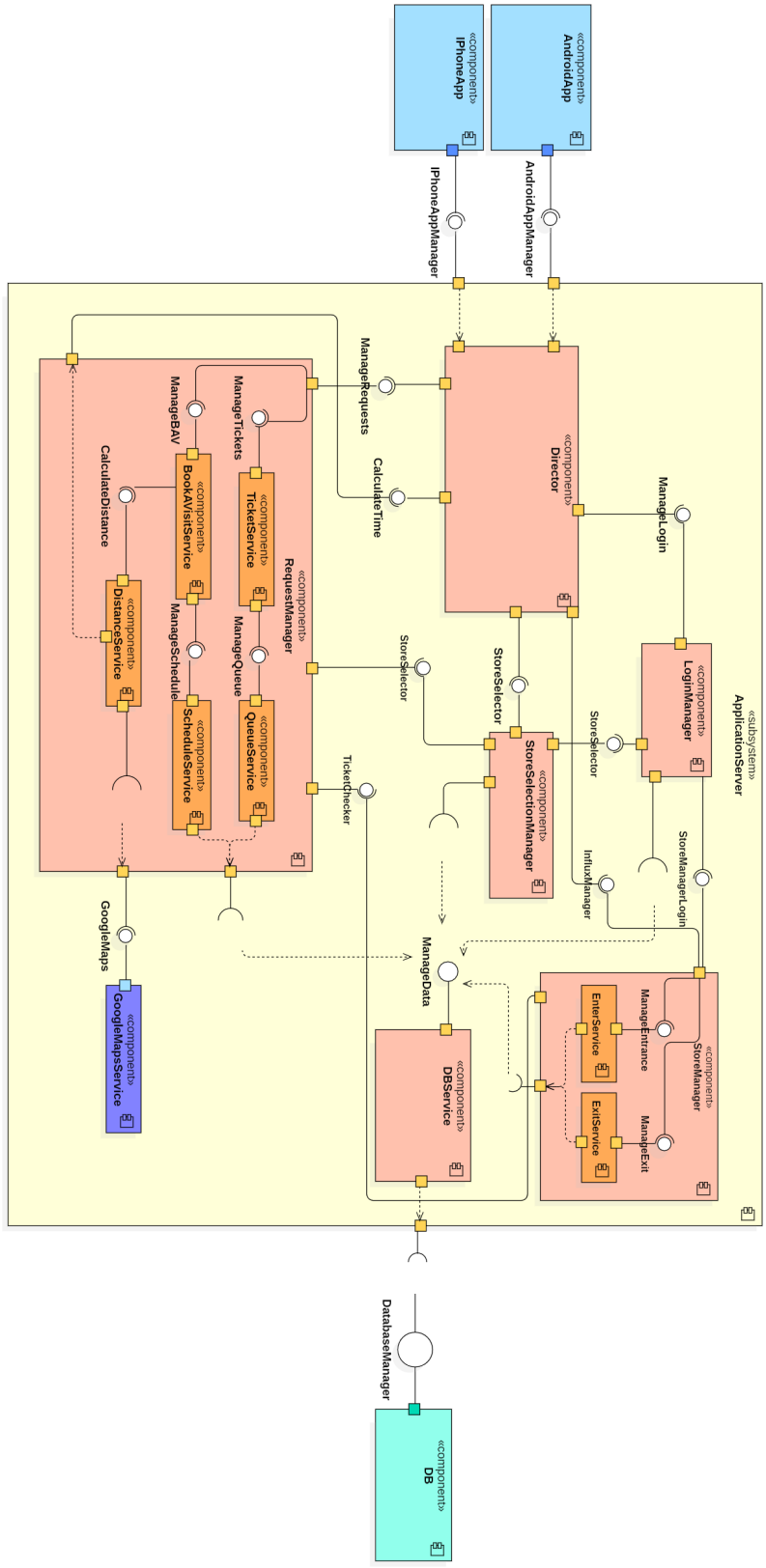

Figure 2: **Main component diagram**

- **ApplicationServer** is one of the three major subsystems that hold the whole application logic of the system. It gathers the input from the Android and iPhone apps, does the calculation, and then transfers the data to the database while taking from it what is needed and getting it back to the phone applications in order to display it to the user. Both PhoneApps and Database are external components that are not part of the application server and the communication to them is done through sockets. The main component of the application server is Director which organizes requests from the phone apps and redirects them to other components that are dedicated to a certain request. Another bigger component is RequestManager which has several subcomponents that are explained in detail further in the text.

- **Director** is in charge of handling requests that are coming from the user via phone app and redirecting them to dedicated components. It is directly connected to the LoginManager, StoreSelectionManager, and RequestManager, and also indirectly connected to the StoreManager and DBService, which leads to Database. Director is also responsible for getting the return information back to the user, such as data from Database, confirmation messages, error messages, and other alerts that might be useful to the user. It is the most important part of the application server because if it fails, no messages or data can go either way.

- **LoginManager** is shown in the simplified form due to its simplicity, as only store managers of the stores will be using this component, since users do not have accounts and therefore do not need to login. It checks the input data username and password with the corresponding data in Database and is responsible for logging in a store manager into their store's account if the credentials are valid. It is then connected to the StoreManager via StoreManagerLogin interface.

- **StoreSelectionManager** is also shown in simplified form since it only features a single information that is a certain store with its name and address. When the selected store's ID is found in Database it is transferred to RequestManager which then handles its requests and retrieves data from Database based on the ID. Because of that, to emphasize the connection of those two components, another socket has been added which directly connects StoreSelectionManager to RequestManager.

- **RequestManager** is the most complex component of the subsystem as it has five additional subcomponents. Two main functions of the application are "Request a ticket" and "Book a visit", which have their own respective services inside RequestManager. Depending on the request, QueueService and ScheduleService are activated, which then communicate with Database to send or gather additional information about the future or the current situation in the certain store (which is known via StoreSelectionManager). If a "Book a visit" request is received, and location services are enabled on the user's device, additional service called DistanceService is activated. It generates a request to the external component GoogleMapsService in order to calculate the distance from the user to the store and get the estimated time needed for the user to get there in order to arrive at a proper time in the schedule. Calculated data does not go to Database but rather goes directly back to Director which then forwards the data to the user, since this information is based on current variables and is not something that can be reused by other users, so there is no need to put it inside the database.
- **StoreManager** is a component that is only used when a store manager of a certain store logs in to their account. From there, they can control the influx of people to the store, scanning and validating tickets as well as sending a signal to Database whenever a customer exits the store. It contains two services, EnterService and ExitService. EnterService concludes whether a scanned ticket is valid or not, and ExitService sends the signal to Database when a customer exits the store and also keeps track of the customer limit in a store.
- **DBService** is a service that does all the communication with Database. It sends all the data requests to DatabaseManager, which then access Database, and then receives the requested information back in order to return it to other managers within ApplicationServer. It basically does the same work as Director component, but talking to the database server subsystem rather than the phone app subsystem.

2.3 Deployment view

In this section we'll touch on the architecture of the whole system in more detail, which is demonstrated with the deployment diagram. The most important components are shown as artifacts and nodes. They represent the physical deployment of software artifacts. Physical hardware is represented by nodes (cuboid) while software is represented by artifacts (rectangles with sheet icon in the upper right corner). We're also following three-layer design in this diagram, with presentation layer being on the left, application layer in the middle, and data layer on the right.

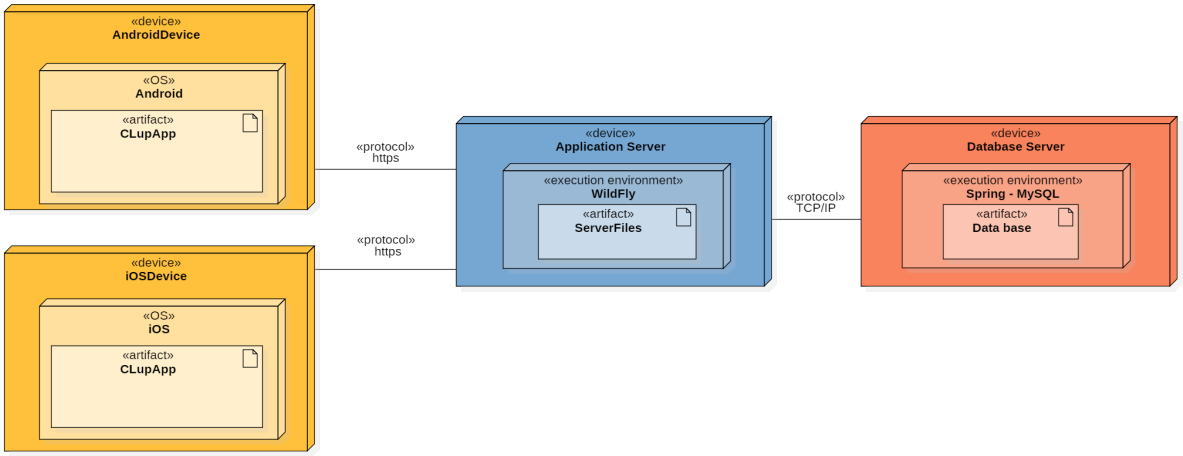

Figure 3: **Deployment diagram - System architecture**

Presentation layer is consisting of two nodes that represent Android and iOS devices. Notice that there is no computer device which is usually common, since there is no version on the app that can be run natively on the computer or through a browser.

Application layer is consisting a single note which represents an application, where most of the computation and application logic is being done. This layer is connected to the presentation layer via HTTPS protocol, and via TCP/IP to the data layer.

Data layer consists of a single node which represents database server. This node is not directly connected to the presentation layer and can only communicate with the application layer.

Nodes and artifacts explanations take on certain components in a bit more detailed manner. Explanations can be found in next part of this section.

Android and iOS devices are both supported by CLup app, unlike the computer version, which is nonexistent due to the ticket scanning system being only available on portable devices (like tablets and phones). These platforms have operating systems that natively support running of the CLup app. The communication with the application server is being done over HTTPS protocol, which is a secure version of the HTTP protocol, over which most of the Internet traffic is going through. HTTPS is fast and very simple, allowing app requests to hastily and securely arrive to their destination.

CLup App is an application that is being run on of the two previously mentioned systems. It is an artifact that should have no functional differences between two versions. Even though the language and the way the app is made and deployed on these devices is different, they should offer exactly the same customer experience.

Application server is a hardware node that is used to run the application part of the system. Our recommended execution environment is WildFly, which is a Java EE application server that consists of the application logic and components mentioned in the previous chapter. As an artifact, it has server files that are needed to run it and connect it to different parts of the system. As mentioned before, it uses TCP/IP to connect and share data with Database server.

Database server is a hardware node that is used to run the database needed for storing all the necessary information about the application. Desired technology to use is Spring Boot with MySQL database, which allows for simple and effective data management and delivery. Database communicates with the application server using JDBC API.

2.4 Runtime view

In the following section sequence diagrams that represent use cases from RASD are shown and explained in detail. Since many use cases are connected and can be placed on the same diagram to better showcase the connection between the components and flow, we have created only three diagrams to represent nine use cases. Also, we have only touched those parts of the system that have to do with the application and the system itself, not displaying real-life scenarios that don't use application.

Components have the same color as in the **main component diagram** which should make things easier to follow.

2.4.1 Book a visit

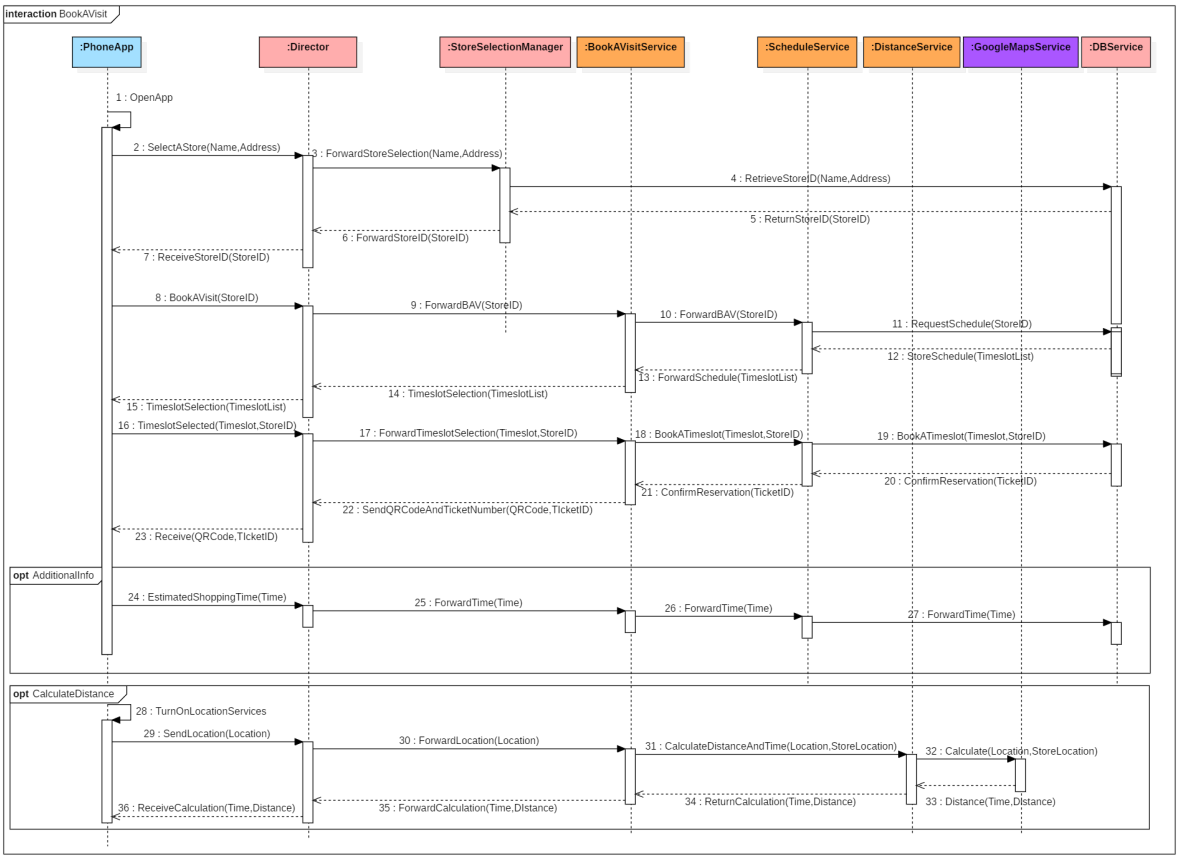

Figure 4: **Sequence diagram 1 – Book a visit**

This diagram consists of the following components that are represented with a lifeline: PhoneApp, Director, StoreSelectionManager, RequestManager (which is represented with its services BookAVisitService, ScheduleService, and DistanceService), GoogleMapsService, and DBService.

The diagram shows one of the basic functions of the app, which is booking a visit to the store at a certain time and optionally, adding estimated shopping time and calculating distance from the store.

Every diagram including this one starts with the app opening and store selection. Store is selected by choosing a store from a list which contains store's name and address. This selection is forwarded through Director to StoreSelectionManager, which then asks for the StoreID and details from the DBService. Every request that comes to the DBService is forwarded to Database which is not shown in these diagrams. After fetching the data from Database, DBService returns the StoreID which goes all the way back the same way to the PhoneApp.

The user then proceeds with using Book a visit feature. The feature is requested along with a StoreID all the way to DBService, using Director, BookAVisitService, and ScheduleService, with the latter two being a part of RequestManager which then retrieves a schedule for that specific store with a list of available timeslots. User then receives the list and picks his timeslot, which starts the return to Database.

After DBService has gotten confirmation and a ticket from Database, it transfers the confirmation along with the QR code and a ticket back to the PhoneApp.

There are additional two actions that the user can perform but doesn't have to. Both actions aim to improve user experience and better approximate waiting time.

One of the actions is user entering estimated shopping time, which is then forwarded directly to database. Database stores the data and uses it when other users request a ticket, in order to approximate their waiting time.

No additional confirmations are returned to the user, because they are not needed.

The second optional action is calculating distance from the user to the store. The system is capable of calculating both distance in meters and in time it takes to get there, using the Google Maps API. If the user has enabled location services and chosen this option, request is sent to BookAVisitService, which forwards it then to DistanceService and GoogleMapsService. After Google Maps API has calculated the distance, it is returned immediately to the user, without going to the database, since this information is used only one time for a specific user, and it is not reusable, which makes it unnecessary to hold in the database.

2.4.2 Retrieve a number

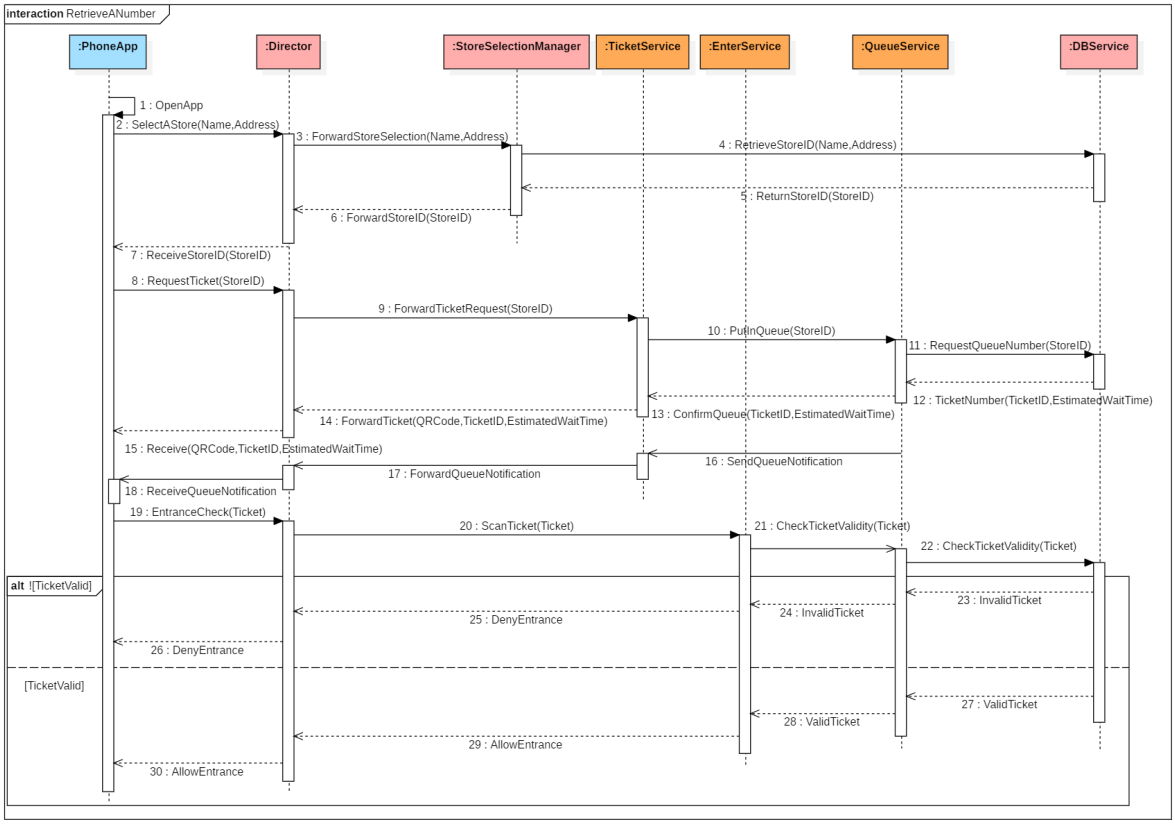

Figure 5: **Sequence diagram 2 – Retrieve a number**

This diagram consists of the following components that are represented with a lifeline: PhoneApp, Director, StoreSelectionManager, RequestManager (which is represented with its services TicketService and Queue Service), StoreManager (which is represented with its service EnterService), and DBService.

The diagram shows one of the basic functions of the app, which is requesting a ticket for entering the store immediately (or in a very near future depending on the queue), as well as the process of ticket validation at the store's entrance.

The process is for the most part very similar to the Book a visit diagram above but uses some different services. Also, the latter part of the diagram can also be applied to the Book a visit diagram, but it has been omitted from there to put the emphasis on different components and communication between them. However, everything from the message number 16 to the message number 27 on this diagram is also happening when entering a store after getting the ticket with "Book a visit" feature.

After opening the app, the user selects a store in the same way as in the previous diagram. The request goes through Director and StoreSelectionManager before getting to the DBService which gets the StoreID from Database and returns it to the user.

The process of requesting the ticket starts with a user generated request. The request goes through Director which then forwards it to TicketService. TicketService generates a ticket and forwards it to QueueService which puts it in the queue. DBService updates the current data for the store, and the ticket with a number and a QR code

is then returned to the user. User also receives approximated wait time he is going to have to wait in queue before being able to enter the store.

When the user gets the notification that it is his time to enter the store, he presents the ticket to the store manager, who scans it. The ticket is then being checked by EnterService and QueueService. Depending on the validity of the ticket, the user is allowed or denied entrance to the store.

2.4.3 Store manager login

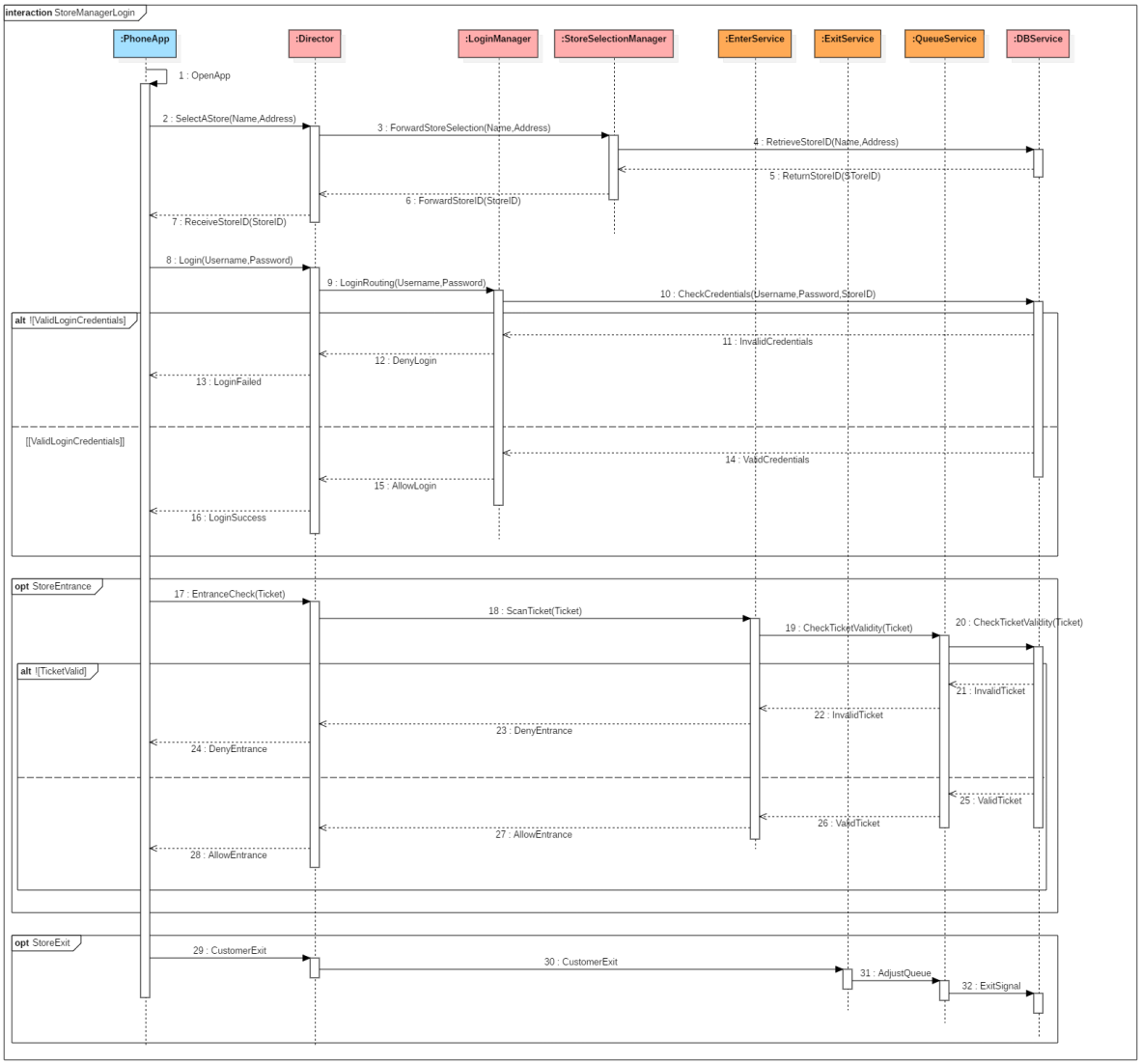

Figure 6: **Sequence diagram 3 – Store manager login**

This diagram consists of the following components that are represented with a lifeline: PhoneApp, Director, LoginManager, StoreSelectionManager, RequestManager (which is represented with its service Queue Service), StoreManager (which is represented with its services EnterService and ExitService), and DBService.

This diagram shows the whole process of the store manager login and app usage when controlling the influx of customers to the store. The diagram consists of three parts – store manager login, customer entrance, and customer exit.

The first part begins with opening the app and selecting a store. The process of selecting a store is exactly the same as in previous two diagrams, so that part will not be explained in detail.

Afterwards comes the store manager login screen. Store manager must provide his username and password (which is given to him after the registration of the store that is done manually by the admins and creators of the app). Request goes through Director and LoginManager all the way to DBService, which checks the credentials in Database, and based on their validity, either allows the login and logs in the store manager or denies the login and returns the error message.

The second part of the diagram begins every time a customer wants to enter the store. Store manager scans their ticket. The ticket information is then sent to EnterService and to QueueService. After checking the ticket validity there and updating Database, the ticket is either valid or invalid, which either allows or denies the entrance to the store for the customer.

The third part of the diagram is much simpler – every time a customer exits the store, the store manager just sends the signal by pressing the button on the app. That signal is then sent to ExitService and to QueueService. The queue and the database are updated. No return signals are given to the store manager after that action.

2.5 Component interfaces

The complexity of developing a mobile application encourages the engineers to split the paramount task into smaller subproblems, or components. Each component is then encapsulating a certain part of the applications function, and communicating with other components through interfaces. The use of interfaces enables the developer of a component to be unaware of the concrete implementation of other components, but just know the syntax of their method calls.

This section gives a thorough overview of the systems interfaces, divided by components they belong to. It is important to note that the interfaces and their methods proposed in this section, do not necessarily represent the exact written counterparts in the implementation, but offer a basic guideline of the component communication.

General interfaces:

- AndroidAppManager
- iPhoneAppManager
- RequestTicket
- EntranceCheck
- GiveTicket
- InformUserToEnter
- CheckTicket
- CalculateTime
- ManageRequests
- StoreSelector
- InfluxManager
- ManageLogin
- ManageData
- DatabaseManager
- GoogleMaps
- ManageTickets
- ManageBAV
- CalculateDistance
- ManageSchedule
- ManageQueue

- StoreManagerLogin
- ManageEntrance
- ManageExit
- TicketChecker

To begin with, the application on the users phone communicates with the system through AndroidAppManager and IPhoneAppManager interfaces, according to the type of the users OS. They enable communication between ApplicationServer, or more specifically, Director component, and the smartphone application. Depending on the type of the user, the interface offers all methods for the interaction with the system.

For example, upon the push of a button in the UI, the application invokes RequestTicket method on Director to propagate the request. Similar to that, an application of a store manager can invoke CheckTicket method on Director, to propagate the specific request to ApplicationServer.

The systems' main global component, ApplicationServer, is divided into smaller core components Director, GoogleMapsService, LoginManager, StoreSelectionManager, DBService, StoreManager and RequestManager. The communication between the core components is also done through interfaces.

For Director to propagate users requests, RequestManager offers methods contained in ManageRequests interface. Other than that, the two components also communicate to calculate wait time through CalculateTime interface.

Furthermore, Director component communicates with LoginManager and StoreSelectionManager components. For LoginManager, it does so through a ManageLogin interface, which enables the propagation of the credential check used when a store manager accesses the application. For StoreSelectionManager, it uses methods offered by StoreSelector interface.

To enable users to choose between the stores using the CLup system, StoreSelectionManager also offers StoreSelection interface, used by LoginManager for store managers and RequestManager for store customers.

The calculation of the distance between the user and the store is done with the help of Google Maps. To implement that, GoogleMapsService offers a GoogleMaps interface to RequestManager. Upon gathering information, RequestManagers subcomponents BookAVisitService and DistanceService perform the distance calculation through CalculateDistance interface.

Depending on the type of the user request, RequestManager further branches into different subcomponents. TicketService and QueueService encapsulate functions needed to perform the get in a virtual line function, and BookAVisitService and ScheduleService serve to carry out the "Book a visit" function of the system. For communication between Director and RequestManager, the former uses ManageTickets and ManageQueue interfaces, whereas the latter uses ManageBAV and ManageSchedule interfaces.

To physically manage store entrances and exits, the CLup system also offers a specific interface for the store managers. The logic behind that interface is contained in

StoreManager component. The component communicates with LoginManager through StoreManagerLogin interface, to execute logins of the store managers. This design choice improves upgradeability for future requests and different users. Furthermore, StoreManager component also communicates with DBService to persist data through ManageData interface. Before persisting, the data is handled by the subcomponents EnterService and ExitService through ManageEntrance and ManageExit interfaces. To connect the full circle of the system function, through TicketChecker interface, offered by RequestManager, StoreManager can control the influx of the customers.

Lastly, in order to persist the applications data, DatabaseManager interface is used. The interface enables the application server, or more precisely, DBService component, to invoke methods to write data to, or read data from, Database. DBService itself, offers ManageData interface to all other subcomponents of ApplicationServer to connect them to Database.

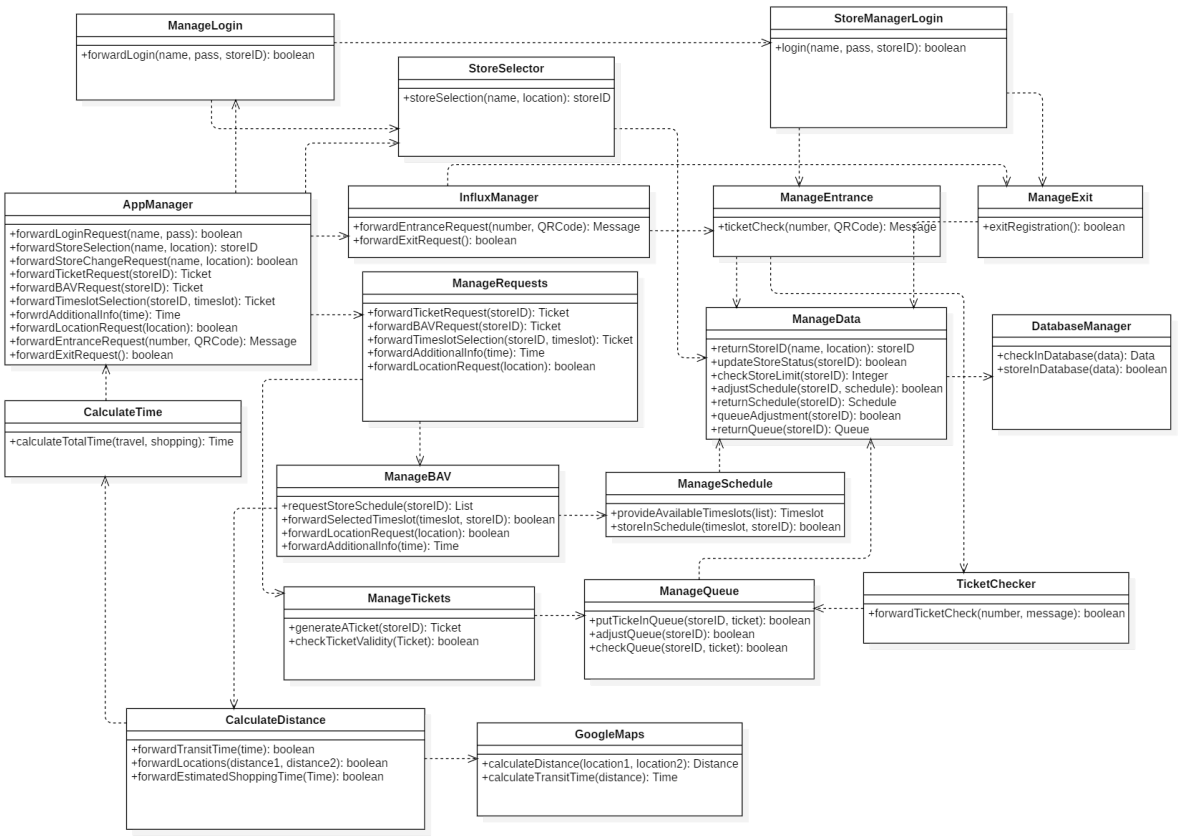

Figure 7: **Interface diagram**

2.6 Selected architectural styles and patterns

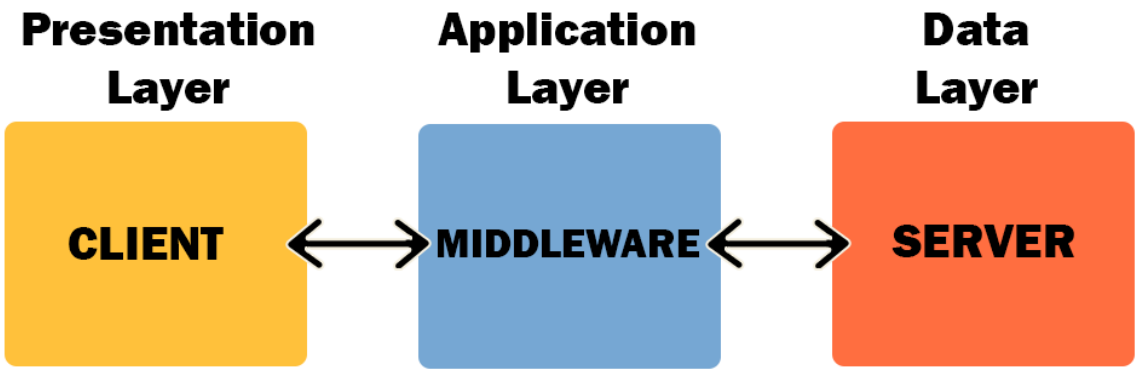

Figure 8: **Three-layer system architecture**

To develop the full system, a three-tier client-server architecture will be used. The choice is encouraged by the popularity of the model, and its valuable aspects such as modularity, scalability or abstraction. The three layers in the three-tier architecture are: presentation layer, application layer, and data layer. By dividing system artifacts in three different layers and enabling mutual communication with abstract interfaces, we hide unnecessary information and improve testability of each single layer. To communicate, the server waits for the clients requests, and upon a request, it extracts data from the database, processes it, and serves it to the client.

Depending on the needs of the system and the work done locally on the client, the client can be thin or thick. For the CLup system, a thin client design is a much better fit, since the calculations can easily be done on the server, and the application has to be usable by all smartphones, even the older ones. The data processed by the server can then be presented to the client device.

Basic software design practices such as modularity and the use interfaces will help greatly with the development, testing and further improvement of the application. Dividing the code in concise modules encourages reusability, and enables extensive upgradeability. Additionally, having functional modules makes component testing possible and quick. Abstraction in code, done through the use of interfaces enhances all the aforementioned effects and relieves the developer of the knowledge and complexity of all components outside his scope.

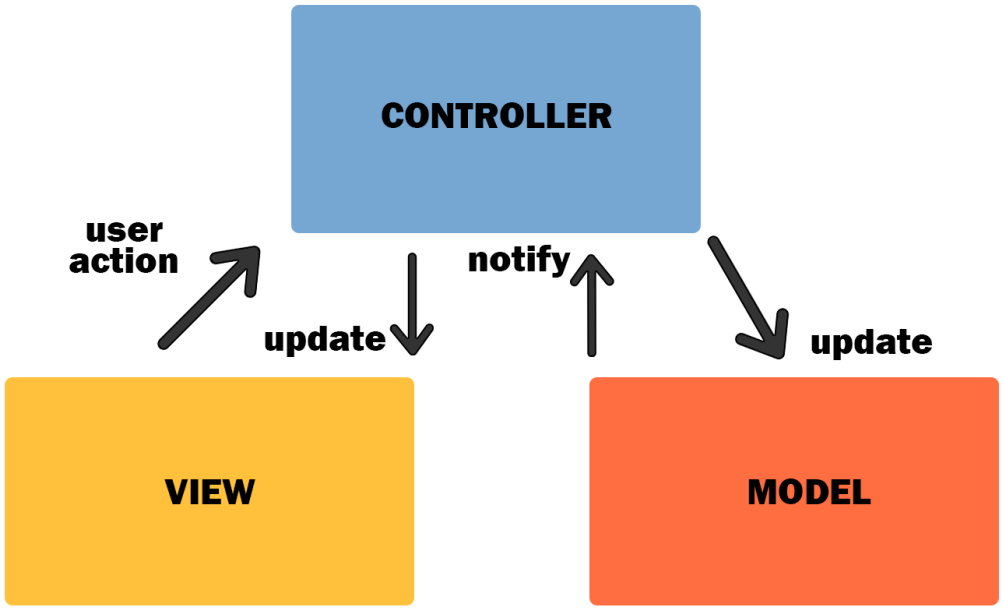

Figure 9: **Model-view-controller**

We would also like to take advantage of the MVC (Model-view-controller) global design pattern. The pattern is widespread in the software engineering community because of its scalability and simplicity, and provides a perfect base for our application. The model component is the main component of the pattern in charge of the business logic. The view component controls the user interface, and the way in which information is represented. And the controller component acts as a bridge between the model and the view. It accepts data from, and creates instructions for the model or the view.

To persist data like store enter and exit times, the system must be able to communicate with a database. DAO (Data Access Object) design pattern acts as an interface and makes that possible. For example, if written in the Java language, JDBC API will be used as DAO.

Main communication protocol used in the system is HTTPS. It provides a simple, popular, and secure connection for message exchange. It is also important to note that to use third-party software, we need to use specific protocols. In the case of GoogleMaps, we need to use the REST API.

In the end, the main software design patterns will be mentioned. Firstly, observer/listener behavioral pattern enables the system to subscribe to the updates of the client devices location and upon change execute new calculations. The pattern defines a one-to-many connection between objects and updates them automatically upon notification. And lastly, bridge structural pattern is used in various places to decouple an abstraction from the implementation. In that way, components can be changed and substituted by new ones completely independently. Through aggregation and encapsulation the bridge separates responsibilities into different classes.

2.7 Other design decisions

The CLup system relies on two main third party components to function properly: Database and the GoogleMaps API. Since most popular database vendors nowadays offer very similar functionalities, the database choice is left to the development team. For the needs of the system, even a NoSQL database system, like Firebase, would work. Firebase is a platform for creating applications, both web and mobile. It was founded in 2011 and is now the primary database option for application development offered by Google. It offers a real-time database, simple interfaces, and high-level security, which makes it well suited for our needs.

To calculate the expected time for a customer to get to the store, the best choice is GoogleMaps API. Through the years, Google polished their map system offer and today they have a widespread, highly accurate, and easy-to-use satellite imagery and street maps. It also offers, real-time traffic conditions and route planning for traveling by foot, car, and public transport which makes it a perfect match for the needs of the CLup system.

In the end, one important design decision should be noted: to avoid security concerns with storing sensible user data in the database, the first version of the system evades user registration altogether. The scope of the system, so far, does not require the system to store such data. However, if the scope of the system grows in such a way that user registration becomes imminent, upgrade of the system built in line with this design document should not be a problem, especially since a form of user registration should already be implemented on the store manager's side.

3 User Interface Design

3.1 Overview

In the following section, mockups for the graphical user interface (GUI) are presented. Some basic mockups can be seen in RASD document, with a few basic screen concepts being presented for both Android and iOS app versions.

In this document mostly one version of the app will be shown since they are nearly identical, with the only differences being due to native button placements on respective operating systems, which are not related to the app.

The base design and colours are taken from RASD. Several new menu screens have been added and some old ones have been modified.
The main aim of the design was simplicity and ease of use. Since we want to make this app available to as many customers as possible, it is important that even people who do not have a lot of experience with different apps have no trouble using it.

Every button has been made bigger and the size of letters is larger than usually in other apps. There is no excess information at the screen at any time and the navigation is rather simple and logical.

Minimal amount of design elements has been used in order to make the app look a little bit more stylish without sacrificing efficiency. Besides the base ones, colours have solely been used to showcase validity of a certain action.

There is also a possibility of adding colorblind mode and a help screen, which will show and explain basic functions of the application and how to use them. This would be highly recommended when implementing the app.

In the text we will mostly be referring to the iPhone devices, although the app is supposed to work on all iOS supported devices. This is due to the fact that people will most likely use this app on iPhone devices and not on iPad and iPod devices. Terms iOS and iPhone could for that reason be used interchangeably in the further text.

3.2 Proposed design

- Already proposed icon look for the app on both Android and iOS devices featured in RASD.

Figure 10: **Android icon concept**

Figure 11: **iPhone icon concept**

- Already proposed title screen concepts featured in RASD. First step when entering the app is selecting the desired store.

Figure 12: **Android home screen**

Figure 13: **iPhone home screen**

Following proposed concepts are shown only on Android devices, however, just as already mentioned, they are practically identical to the iPhone version.

- Main screen selection of the app, featuring selection for all most vital functions of the app.

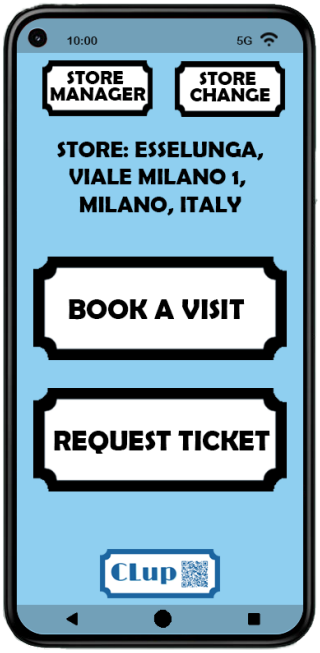

Figure 14: **App main screen**

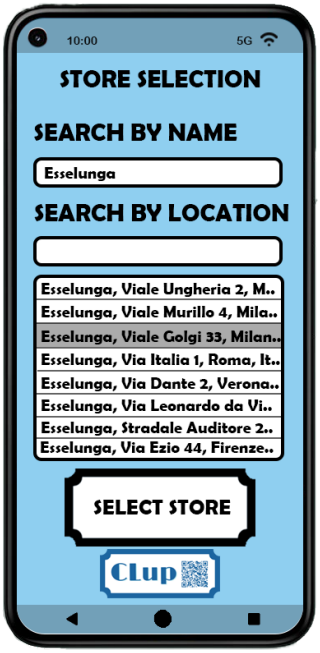

Figure 15: **App store selection**

- Login screen for the store manager and generated ticket screen.

Figure 16: **App store manager login screen**

Figure 17: **App ticket request screen**

- "Book a visit" feature has an implemented calendar for date selection and a list of available timeslots for a selected date in a certain store.

Figure 18: **App "Book a visit" screen**

- Additional info screen when booking a visit. A user can enter estimated shopping time, and if their location is on, Google Maps will estimate the distance to the store and calculate the total time it takes to get there and do the shopping.

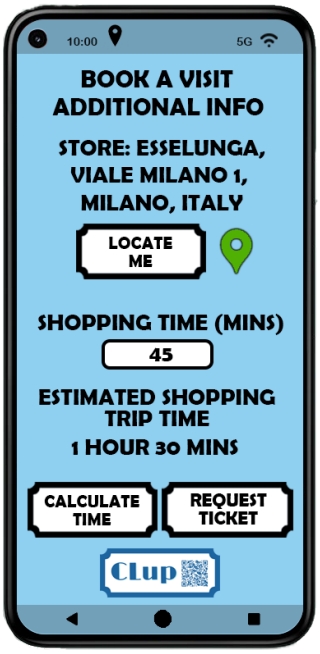

Figure 19: **App additional info screen (location on)**

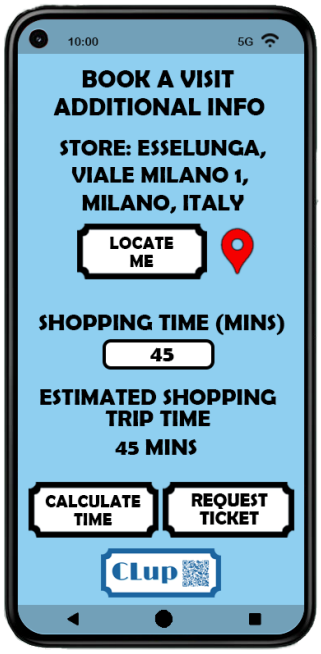

Figure 20: **App additional info screen (location off)**

- Screen featured when a store manager is logged in. Both examples of a valid ticket scan and an invalid ticket scan are shown.

Figure 21: **App store manager screen (valid scan)**

Figure 22: **App store manager screen (invalid scan)**

4 Requirements Traceability

To further clarify and reason the implementation components proposed in the **main component diagram** this chapter connects them with the goals and requirements, specified in the RASD document. Each goal **[Gn]** will be presented, and connected with the mapped requirements **[Rn]** and responsible design counterparts.

R1	The user must be able to select a specific store in which they want to do the shopping.
R2	The user must be able to request a number and a ticket.
R3	The user must be able to receive a number and a ticket.
R4	The user must be able to physically retrieve a ticket from the printer containing a number and a QR code.
R5	A new ticket must be printed whenever a user physically retrieves the old one.
R6	The store manager must be able to scan a QR code.
R7	The store manager must be informed by the application if a user tries to enter the store out of order.
R8	The store manager must be informed when the capacity of the store is full.
R9	The store manager must be able to alert the system whenever a customer exits the store.
R10	The store manager must be provided with the login credentials upon request to the system administrator.
R11	Allow the user to receive a precise estimation of waiting time when retrieving a number.
R12	The system must calculate an estimation of the waiting time based on data.
R13	The system must be able to update its estimated waiting time in real time.
R14	The system must be able to send an update to the user in specific intervals regarding estimated waiting time until it's their turn.
R15	The user must be able to request to see all the available timeslots in that specific store.
R16	The system must be able to provide the user with the list of all available timeslots upon the request.
R17	The user must be able to select a specific timeslot.
R18	The user must be able to receive a confirmation of his timeslot reservation, along with a number and a ticket.
R19	Allow the user to be at most five minutes late for his reservation before cancelling his ticket.
R20	The user must be able to specify expected duration of his visit to the store.

Table 1: **Requirements table**

G1.1	Allow the user to retrieve a number through the application. Requirements: R1, R2, R3
	Components: AndroidApp/IPhoneApp Director StoreSelectionManager RequestManager TicketService QueueService ScheduleService DBService and DB

Table 2: **Components table - Goal 1.1**

G1.2	Allow the user to retrieve a number physically from the printer. Requirements: R4, R5
	Components: Director StoreSelectionManager RequestManager TicketService QueueService ScheduleService DBService and DB

Table 3: **Components table - Goal 1.2**

G2	Allow the store manager to control the entrance of customers via QR code scanning. Requirements: R6, R7, R8, R9, R10
	Components: AndroidApp/IPhoneApp Director LoginManager StoreSelectionManager StoreManager EnterService ExitService RequestManager DBService and DB

Table 4: **Components table - Goal 2**

G3	Allow the user to receive precise calculations of the waiting time. Requirements: R11, R12
	Components: AndroidApp/IPhoneApp Director RequestManager DistanceService GoogleMapsService DBService and DB

Table 5: **Components table - Goal 3**

G4	Allow the user to be updated on the store waiting time situation. Requirements: R13, R14
	Components: AndroidApp/IPhoneApp Director RequestManager GoogleMapsService DBService and DB

Table 6: **Components table - Goal 4**

G5	Allow the user to "book a visit" to the store. Requirements: R15, R16, R17, R18, R19, R20
	Components: AndroidApp/IPhoneApp Director StoreSelectionManager RequestManager BookAVisitService TicketService QueueService ScheduleService DBService and DB

Table 7: **Components table - Goal 5**

5 Implementation, Integration and Test Plan

5.1 Overview

In this section the whole process and the idea of the system implementation is explained in detail. The whole integration and implementation process of the already presented subcomponents of the system, together with the testing plan is a very detailed and specific process. In order to get the best possible picture of the recommended implementation process, both diagrams and detailed explanations are used.

As mentioned in the second section, the system is divided into three layers: presentation layer, application layer, and data layer. Each of these layers holds one major subcomponent; Phone app, Application server, and Database, respectively.

These subcomponents feature many different services, managers, and interfaces, which can be found in the second section as well. Detailed implementation of those is available only for the Application server subcomponent since most of the application logic is happening there.

Google Maps API and Database are external and somewhat outside of our control, which also has to be taken into account during the integration and testing process.

5.1.1 Importance of features

Function	**Level of importance**	**Implementation difficulty**
Login	Users – Low; Stores - High	Low
Requesting a ticket	High	Medium
Booking a visit	Low	High
Locating services	Low	Medium
Estimated time calculation	Low	Medium
Store selection	High	Low

Table 8: **Importance of features**

5.2 Implementation

The desired implementation process technique for this project is top-down. Since all major system components are already known and the application has a clear cut of what it should exactly do, we believe this is the way to go. By using this design technique, most of the biggest components are implemented and defined at the beginning and we work our way down to the smaller ones. This gives us a clear picture of the whole system and strong points to lean on when implementing smaller parts. The opposite technique bottom-up, which is used to build the system from the ground up, defining smaller components and then searching for the bigger picture, would be slower to implement and unnecessary since the main goals and components are clearly defined.

The order of developing and integrating major components of the system should be the following:

First we develop a basic app and set up a database – this way we already have a basic system and can work on connecting it with an application server for basic functions.

The setup of the application server and application logic follows.

Director, RequestManager, and DB service should follow, creating a set of the most important subcomponents that allow the app to communicate with Database. Smaller parts of these components, like QueueManager, TicketManager, and others are to be implemented at the end.

At the end, other smaller components, like StoreSelectionManager, StoreManager and LoginManager are implemented, to extend the functionalities of the application. After that other components of the RequestManager come in line.

Finally, GoogleMapsService can be implemented at the end since it is used the least and is the least important part of the system.

This design flow will allow for easier testing of the major components. Testing interconnection and communication within the components will be of key to develop a reliable system. Once that is set up, smaller functionalities of the system can be implemented and tested, with the focus staying on the bigger and more important functions. That way the system can even go through some sort of a beta phase testing with the most important functionalities being set up properly and having the customers test them out. Even if some components are not tested and set up then, the app can go into circulation with more important functions working, allowing further development to not slow down the release of the app.

5.3 Integration strategy

Using the following components, the strategy regarding integration order explained in the previous section is showcased. Each component is implemented one-by-one or in groups of up to three subcomponents. This policy allows fastest development, while still maintaining a coherent global picture in mind.

After PhoneApp and Database have been set up, the ApplicationServer is being implemented.

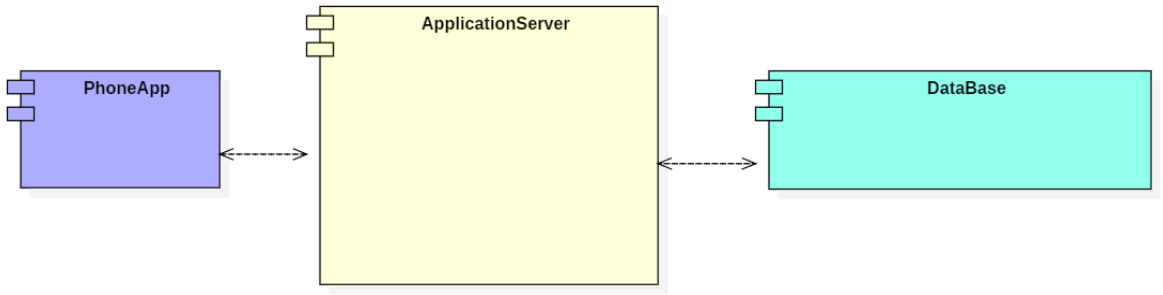

Figure 23: **Integration diagram 1**

Director and DBService come first – ensuring that the connection and communication is properly implemented between PhoneApp and Database. Setting the director up is the most important step – with proper basic implementation, all of the other components and services are going to be to easily implemented and connected to Director.

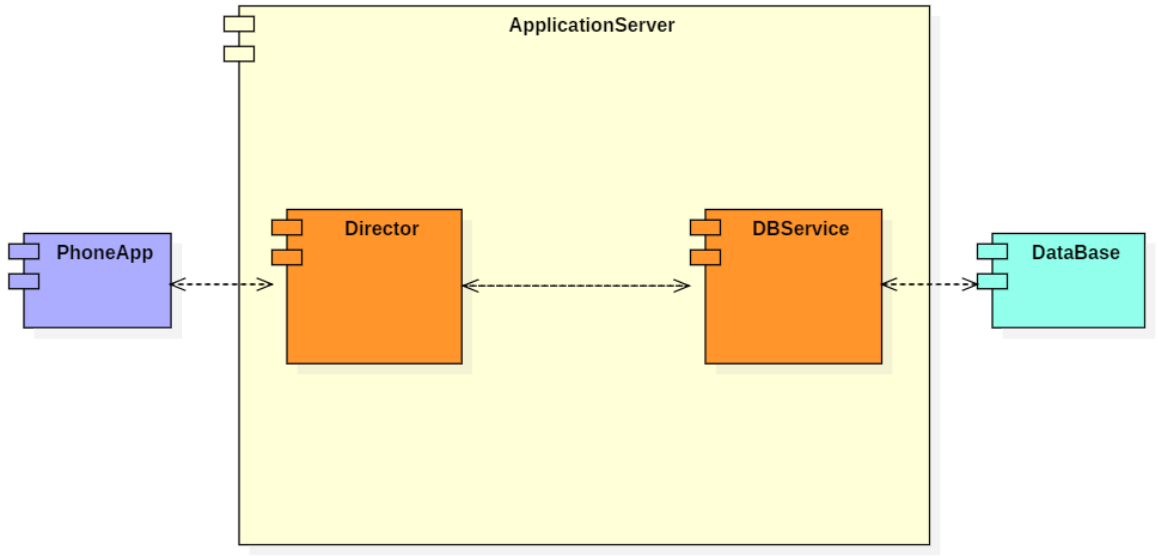

Figure 24: **Integration diagram 2**

Request manager is the third subcomponent to be implemented, being connected both to Director and DBService. Most of the requests will go through RequestManager before heading to DBService, rather than going directly through Director, because some additional computation has to be done via the services located in RequestManager.

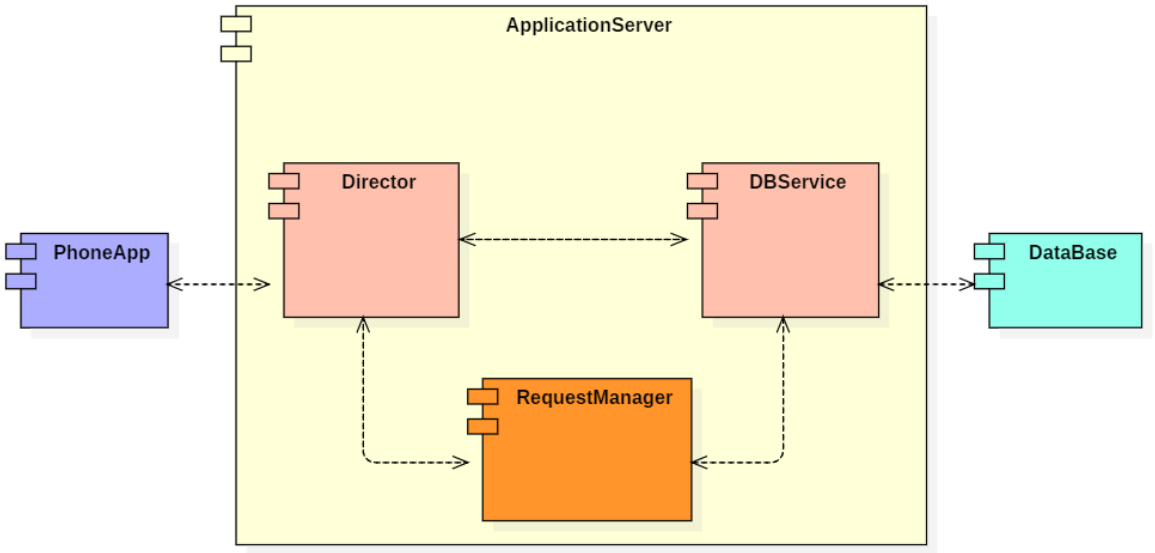

Figure 25: **Integration diagram 3**

Managers regarding store selection and store manager login are implemented. This brings in another part of the app into existence, with all of the other components implemented so far being user oriented, while this one is admin oriented. With this, the store manager connection and login to the store can be tested.

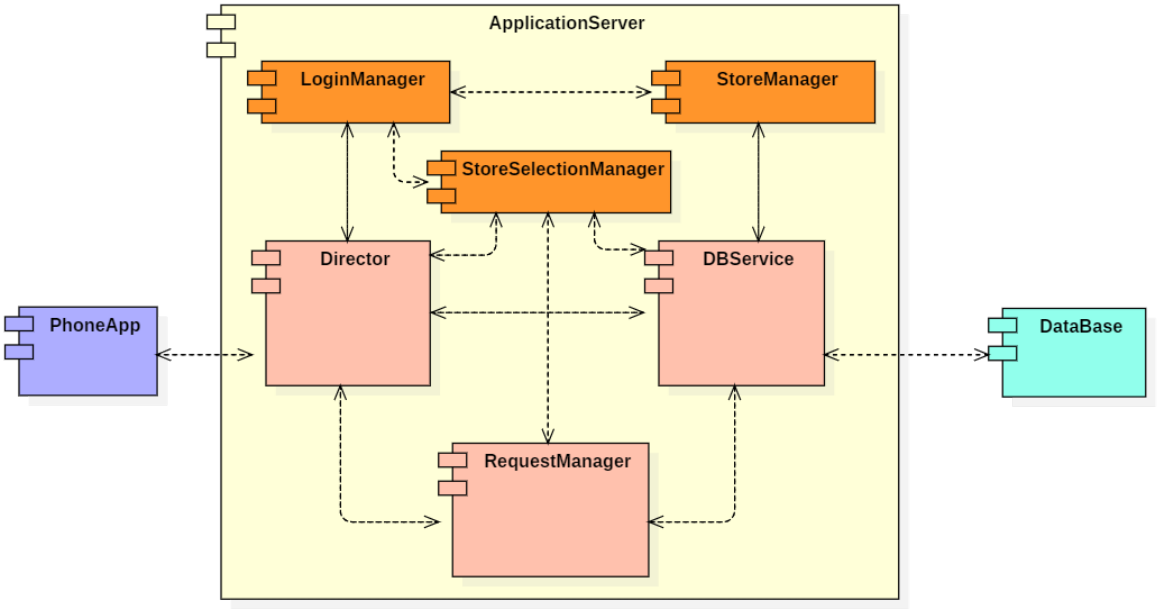

Figure 26: **Integration diagram 4**

Lastly, all of the services in StoreManager and RequestManager are implemented. Now the ticket and queue handling are available, controlling influx of people to the store, and other functionalities like "Book a visit" are available. After all of these services have been set up and properly interconnected, GoogleMapsService can be imple-

mented to finalize the system.

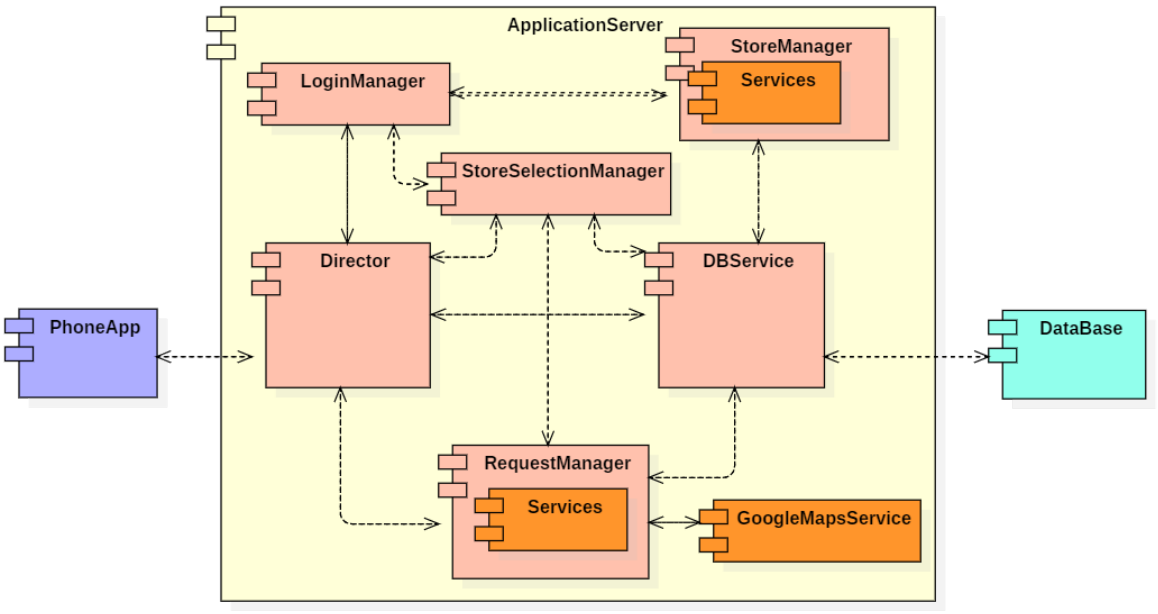

Figure 27: **Integration diagram 5**

5.4 Testing

Designing a good test plan is crucial in application development. It reduces uncertainty upon release, excessive work for error resolving and bug fixing, and as a result, lowers the overall costs of the process. This paramount task usually consists of UI testing, functional testing, and security testing.

Today, there is a lot of available tools for quick and thorough UI testing through the use of automated scripts. Besides that, an application should also be manually tested by few real users. In our case, it is best to have friends and family do that, since they cover almost all demographics.

To check whether all functional requirements are met, we use functional testing. The best method to carry out functional testing is the black box technique. It is a method of software testing that examines the functionality without peering into the applications internal structures.

The last important part of the testing plan is security testing. To test whether the system is vulnerable, we use white box testing. To carry out white box testing, we check the software code for internal security holes, and for broken or poorly structured loops or paths.

After going through with our testing plan, the application is ready for fine tuning and its first release.

Since this application is planned to be used on a large scale, it is impossible to re-create all of the real-life scenarios and different situations that can happen during testing. The plan is to release the app for a selected number of stores only (ex. 10) and test the real-world use for a week. After that, we will be able to get a better view of the possible bugs and errors that can cause the app to not function correctly, and fix them. Since the complexity of the app is not at a very high level, we suspect that most of the major issues will be solved by the end of that first week of testing, and that the app will be more than ready to be released for everyone in a very short amount of time.

6 Effort Spent

Task Description	Time spent[hours]
Introduction	1
Architectural design	8
User interface design	10
Requirements traceability	2
Implementation, integration and test plan	3
Total	24

Table 9: **Effort spent - Robert Medvedec**

Task Description	Time spent[hours]
Introduction	3
Architectural design	7
User interface design	1
Requirements traceability	8
Implementation, integration and test plan	4
Total	23

Table 10: **Effort spent - Toma Sikora**

