Design Document - DD

January 2021

KONG XIANGYI
&
ZHANG YUEDONG

Contents

Chapter 1

INTRODUCTION

1.1 Purpose

The purpose of this document is to explain how we design and build CLup application. We demostrate the architecture design of the system, based on the rule of Top-down design approach, we describe the different design characteristics. We explain it from the view of component, deployment and runtime. To be more specific, we also show the user interface design to illustrate the CLup application step by step. The purpose of interface design is user-friendly and efficiently. Our aim is to let other stakeholders can easily understand and know the structure of CLup application.

1.2 Scope

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- Click Customer : The customer has the required technology to access the store. I.e a smartphone. They can use the customer terminal software.
- Brick Customer : The customer doesn't have the required technology to access the store, they have to hand out "tickets" on the spot.
- Store Manager : They have to manage the Store System, include the software and hardware.

- Ticket: The ticket is a document which contains three key information: QR Code, the estimated departure time, the queue number, and the Store Planned Roadmap. To the click customer, it's **E-ticket** but to the brick customer,it's **Paper Ticket**,and doesn't contain the estimated departure time, and just a General Store Map without the Planned Road.
- QR Code : When customer booked a visit, they will received a QR Code.
- QR Code Scanned Machine : A hardware, the Click Customer can use this machine scan their QR code.
- Tickets Hand-Out Machine : A hardware, the Brick Customer can use it retrieve their Ticket.
- Store Planned Roadmap: A store map that includes a finer way which is recommended form Store System.
- Digital Counterpart : A hardware, it with show the queue number.
- Store Back-End System : A software, as the back-end manages all stuffs.
- On-Time Store Data : A dataset that includes the store's on-time date.
  - The current queue
  - The customers in the store
  - The maximum number of people in the store.
- Long-Term Customers: The Click Customer who visited the store more than one time by the CLup mobile application.

1.3.2 Acronyms

- RASD - Requirement Analysis and Specification Document
- DD - Design Document
- CLup - Customers Line-up
- UI - User Interface
- IOS - iPhone OS

- PC - Personal Computer
- IaaS - Infrastructure as a Service
- CRM - Customer Relationship Management
- LAN - Local Area Network
- RAPS - Reliable Array of Partitioned Service
- RACS - Reliable Array of Cloned Services
- MVC - Model-View-Controller

1.3.3 Abbreviations

-

1.4 Revision history

1.5 Document Structure

Chapter 2

ARCHITECTURAL DESIGN

In this chapter, we will describe the architectural design of our system.

We will use the Top-down design approach, design the very high-level structure first, and then gradually work down to detailed decisions about low-level constructs. Finally, arrive at detailed decisions.[5] Let us start with the High-level components and their interaction.

2.1 Overview: High-level components and their interaction

We chose **3-Tiered architecture** with the **Thin Client** strategy for our system. As shown in Fig.2.1.[1]

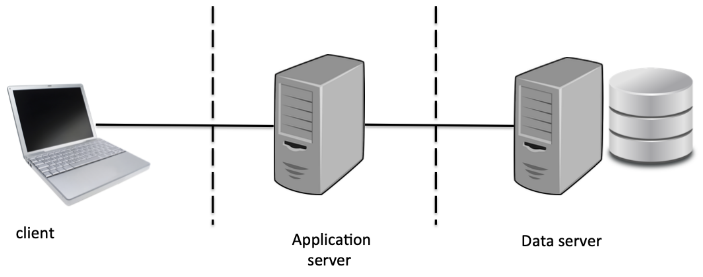


Figure 2.1: 3-Tiered architecture

Tier-1 is the **Presentation Layer**. This layer will deploy the Click Client's Mobile Application, the Store Manager's Management System, and even Digital Counterpart and Ticket Hand-Out Machine's presentation.

Tier-2 is the **Logic Application Layer**. This layer will deploy our Back-End System's components.

Tier-3 is the **Data Access Layer**. This layer includes our DBMS and the Data Base.

Finally, our high-level architecture is shown in Fig.2.2. The Store Manager's PC and other hardware will connect with the Ethernet Switch Hub. The Click Customer's Mobile App will communicate with our Back-End System via the Internet.

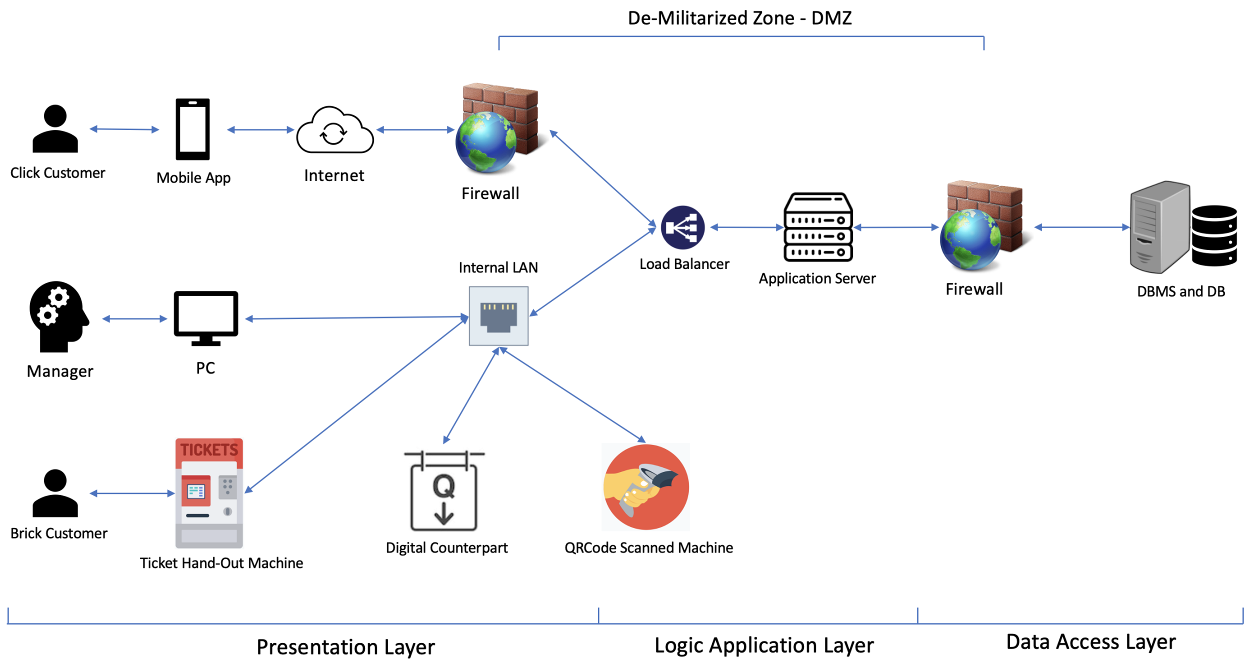


Figure 2.2: High level architecture

More detailed components will be introduced in the next sections. In particular, in Sec.2.3. There, we will introduce the DMZ, load balancer, and other Deployment plan selections.

2.2 Component View

The following diagram Fig.2.3 is the **Component Diagram**, and it contains all components in our system. This diagram shows the 3-Tiered Architecture of our system. The Yellow components are in Presentation Layer. The blue components are in the Logic Application Layer, and the Green components DBMSServer is in the Data Access Layer. We will separately introduce all these components in this section, for the interfaces, please go to Sec.2.5.

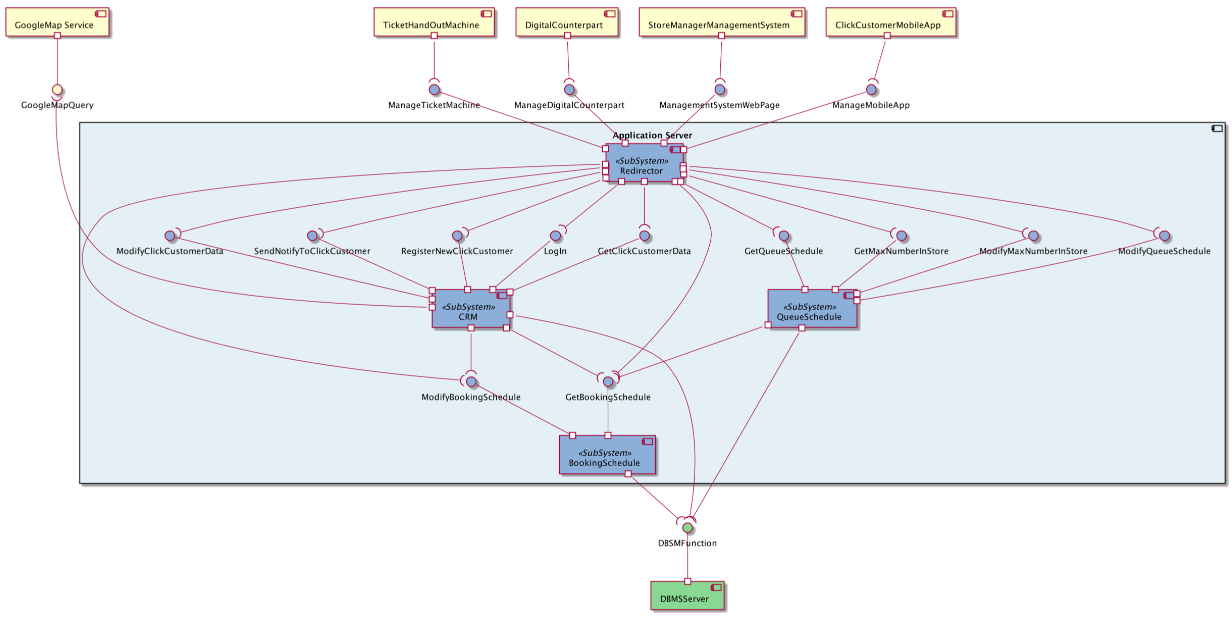


Figure 2.3: Component Diagram

The **Redirector** component is a bridge for communicating the Presentation Layer and the Logic Application Layer. It is transparent to User and back-end systems, plays the role of forwarding information on both sides, and connect interfaces. It has four subsystems, respectively, responsible for communicating with four types of devices, as shown in Fig.2.4.

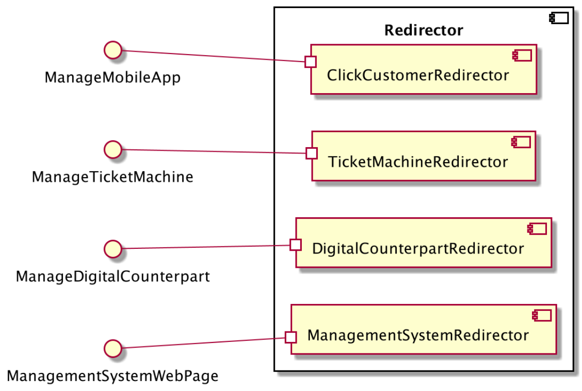
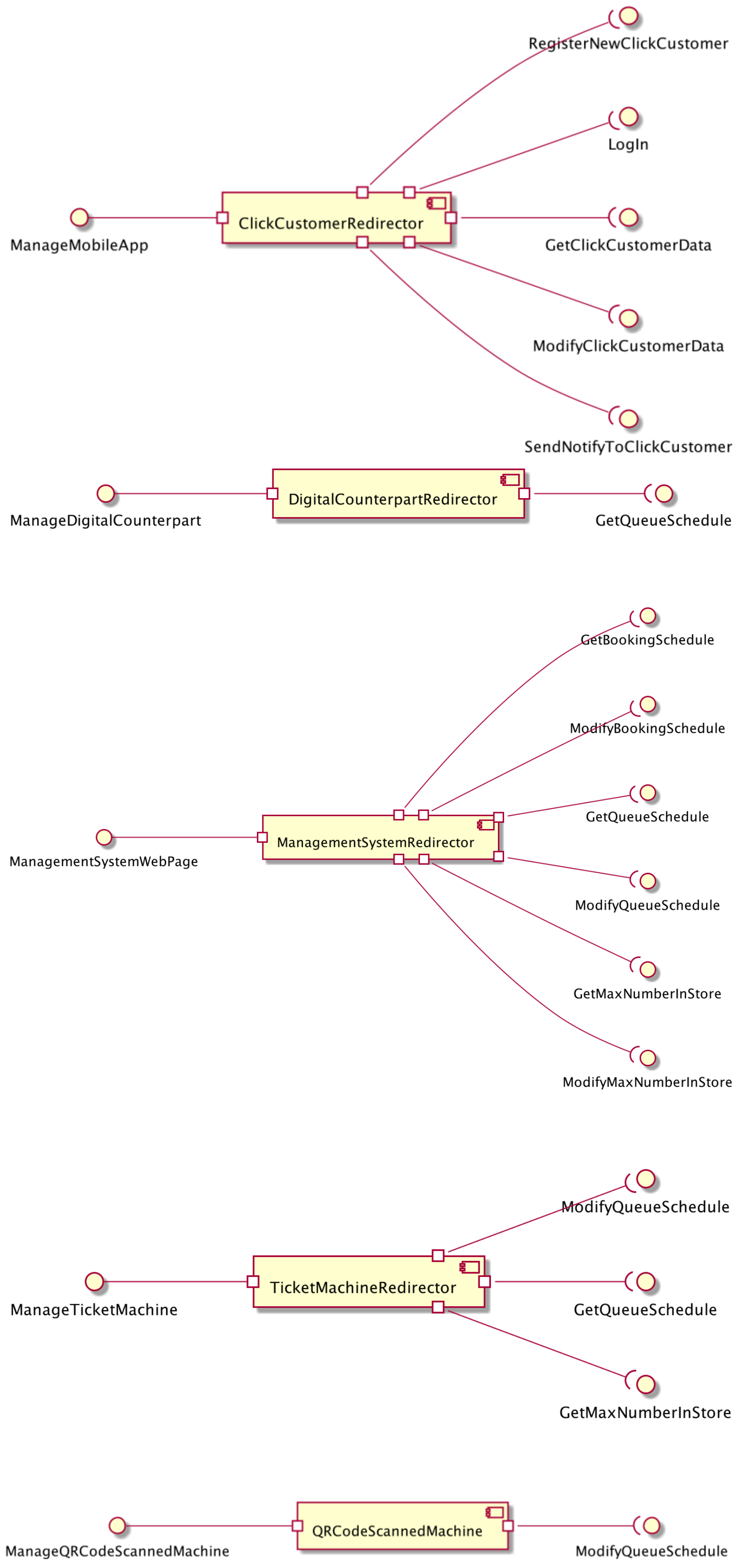


Figure 2.4: Redirector Component Diagram

The **ClickCustomerRedirector** will transfer the message between Mobile Application and the Application Server. The Register message will pass


RegisterNewClickCustomer
LogIn
ManageMobileApp
ClickCustomerRedirector
GetClickCustomerData
ModifyClickCustomerData
SendNotifyToClickCustomer
ManageDigitalCounterpart
DigitalCounterpartRedirector
GetQueueSchedule
GetBookingSchedule
ModifyBookingSchedule
GetQueueSchedule
ManagementSystemWebPage
ManagementSystemRedirector
ModifyQueueSchedule
GetMaxNumberInStore
ModifyMaxNumberInStore
ModifyQueueSchedule
ManageTicketMachine
TicketMachineRedirector
GetQueueSchedule
GetMaxNumberInStore
ManageQRCodeScannedMachine
QRCodeScannedMachine
ModifyQueueSchedule

by the RegisterNewClickCustomer, and look out the Valid/History Booking will use the GetClickCustomerData Interface, this Interface will return all information of this Customer. For the most important operation - Booking a visit, use GetClickCustomerData to get the available Time/Date and the average history duration(if available). Then, through the ModifyClickCustomerData Interface to submit the Booking, by the way, it also can modify other available data of this Customer. Moreover, the SendNotifyToClickCustomer Interface will handle Notify message. The Mobile Application can actively "pull" Notify message through this Interface from the Back-End System. To "push" Notify information, we will use the Observer Pattern. It will introduce in Sec.2.6

The **TicketMachineRedirector** will transfer the Ticket Hand-Out Machine's message. It needs two pieces of information, the allowed maximum number of people in the store and how many Customers in the Queue in time. So The GetQueueSchedule and GetMaxNumberInStore Interface can do these operations, and these two fields will show on the Ticket Machine's screen. Finally, the retrieve ticket operation will handle by the ModifyQueueSchedule Interface. It will put a BrickCustomer into the Queue.

The **DigitalCounterpartRedirector** will help the Digital Counterpart display the next queue number. The Customer can enter the store after found his number on the Digital Counterpart.

The **ManagementSystemRedirector** will function for the Management System of our Store Manager. This redirector will transmit the message between the Presentation layer's web page and the Back-End System. Through the Interface shown in Fig.2.5, the Manager can realize his operation like reschedule someone's Booking, adjust the Queue order, or lower the maximum number.

The **QRCodeScannedMachineRedirector** is working for the Ticket QRCode Scanned Machine. When the Enter Machine scanned someone's QRCode, it will directly use the ModifyQueueSchedule to move this Customer from the Queue list to the CustomerInStore list. If this Customer is not in the Queue list, the ModifyQueueSchedule will not return OK, which means this operation not successful. Of course, the Exit Machine will do similar things, move out the Customer from the CustomerInStore list.

The next three subsystems are the core subsystems in our application layer.

The **CRM** component is responsible for communicating with Click Customers' End, handle the register, Log-In, Booking, data-send, and notify-send operation. The Booking operation will communicate with the Booking Schedule component, get the available data/time, send the Booking Information, generate E-Ticket, and store all data in the database. In addition, it must also help the Customer calculate the estimated departure time from depart place to the store, so it will call the GoogleMap API to query it.

The **Booking Schedule** component is working for all Booking tasks. Receive the Booking from CRM, handle the Store Manager's modified Booking, send the Booking schedule to the Queue Schedule component, and store all Booking Schedule to the database.

The **Queue Schedule** subsystem is the main component to communicate with the Store Manager's Management System. It has to take the Booking schedule to calculate the Queue order and handle the Store Manager's modification for the Maximum value or Queue order.

At last, the **Database System**, it will manage all database, and handle the CRM, BookingSchedule, QueueSchedule's Query/Insert/Update.

2.3 Deployment View

Our deployment situation, as shown in Fig.2.6. Three colors respectively represent the three layers. The Green devices are in the Data Access Layer, the Blue devices are in the Logic Application Layer, and the Yellow devices are in the Presentation Layer.

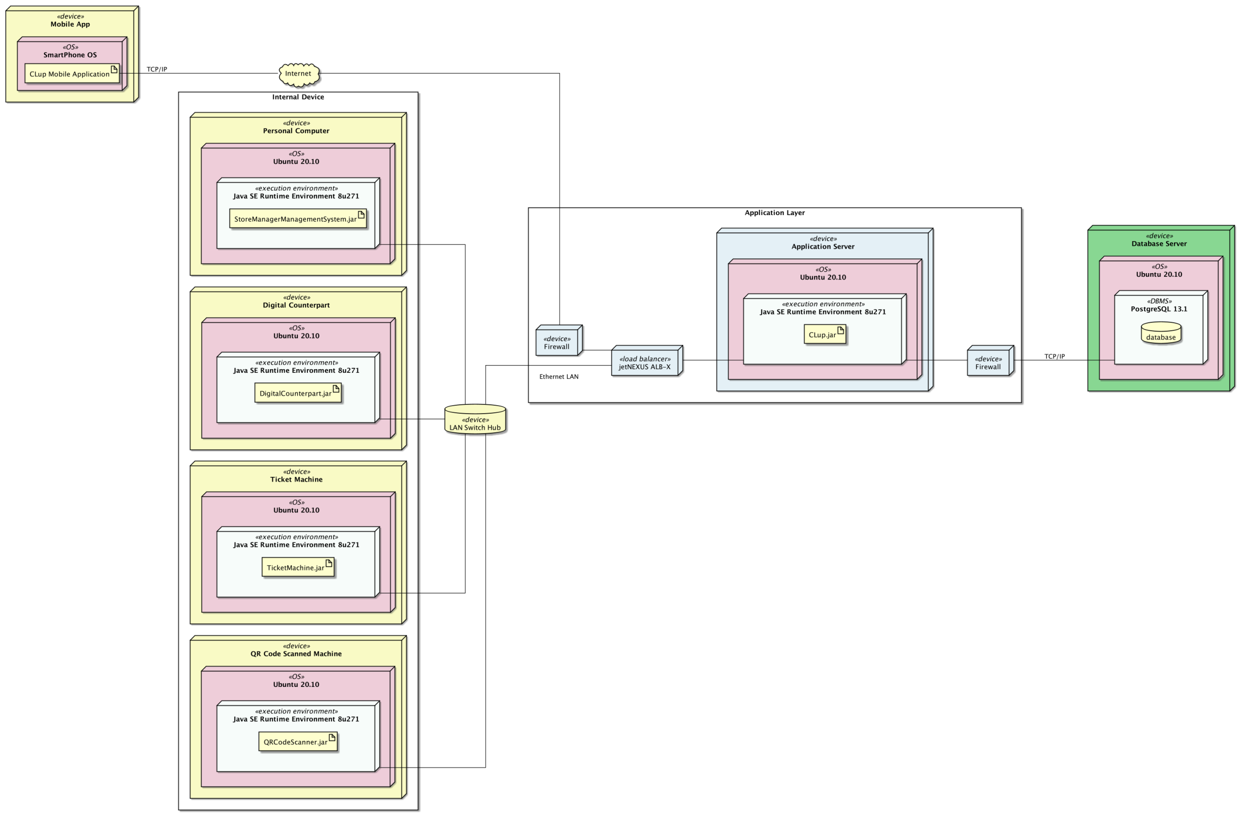


Figure 2.6: Deployment Diagram

For all our devices, to reduce the development/management/maintenance cost, we will use the same Operation System - Ubuntu 20.10, and run in the same Runtime Environment - Java SE Runtime Environment 8u271.

In the **Presentation Layer**, The Store Manager's PC, the Digital Counterpart, the Ticket Machine and the QRCode Scanned Machine are the Internal Devices, which means we must manage them by ourselves. We built a LAN to connecting them with the Application Server via the Switch Hub. In our LAN, we used the Ethernet Protocol. Moreover, we have chosen the **Thin Client** strategy in the system architecture, so all our Internal Devices can run on the Raspberry Pi 4. It is a very low-cost deployment solution.

In contrast, the Mobile Application is the External Device that will run on the Click Customer's SmartPhone. It has to connect with Back-End System via the Internet. We will separately develop the Client End for the two kinds of systems - IOS and Andriod System.

In the **Application Layer**, We refer to the network security part of the information system book P.243. We used two Firewalls to build a **DeMilitarized Zone - DMZ**. The external network can only access the resources exposed in the DMZ (Our Application API), and the rest of the network is behind the second firewall. The DMZ functions as an isolated subnet between the Internet and the private network, So that our database can be better protected.[1]

For the **Load Balancer**, we must first ensure that our Internal Devices can allocate enough Server resources and then ensure the Click Customer's resources, so we considered two plans.

1. Allocate dedicated Application Servers for our Internal Device, and other Application Servers connect with Load Balancer.

2. Do not allocate server resources separately for internal devices, but set higher weights for Internal Devices in the load balancer.

Plan 1 can guarantee that our system's most important part can work well, whether the balancer is working or how crowded the network is. Plan 2 ensures that when some servers are not working, the well-function servers can always allocate to Internal Devices. After weighing, we finally chose plan 2. Because we think Plan 2 has higher system availability/maintainability.

As mentioned above, for the **Application Server**, we adopted the Reliable Array of Cloned Services - RACS, like the figure in the Information System Book P.202[1], Fig.2.7. The load balancer must allocate the higher weights to Internal Devices, and all our Application Server will run the same application. There is a different point in our case from the figure: we have separated the database and the Application Server. Nevertheless, cause our business is "write-intensive", so we used the shared-disk configuration in essence.

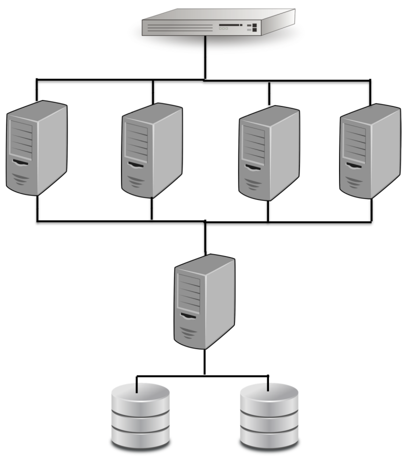


Figure 2.7: RACS: Shared-disk o cluster Configuration

Finally, the **Database system**, we selected PostgreSQL as the Database Management System cause it is Open Sources and stable. Then we will use the relational database to implement it. For Security, as the DMZ mentioned above, only our private network can access this system, so that the external network will difficultly attack our database.

2.4 Runtime View

This section will describe the two most important processes in our system - the Click Customer booking process and Store Manager manage processes.

The **Click Customer booking process** is shown in Fig.2.8. There are three components of our Application Server participated in. The redirector forward all message for two sides, and the CRM system handle all Customer stuffs like the bank windows. When a Customer sends a booking, the CRM system will not generate tickets immediately. Because it must wait for all customers within the closed period to book their visit, and then it can calculate the route map in the Store for each customer according to their wanted list. By the way, there we used the Observer Pattern, which we will introduce in Sec.2.6. However, the CRM will generate the Ticket when someone's booking is changed and then send a notification to the corresponding Customer.

After-all, the Customer can check their Ticket when they received the Notification. If they checked it too early, they would only found their historic Ticket.

The second process is the **Store Manager Management System process**, and it is shown in Fig.2.9. This process presented all functions of the

Store Manager Management System. The four operations in the figure are independent of each other, and there is no sequence. The management system can execute any process according to the operations of the Store Manager. Since the first three operations in the figure are clear and precise enough, we will focus on the fourth operation - Reschedule the booking.

When a Store Manager does the **Rescheduling job**, they have to check the Booking Schedule first, and then choose a range they want to change, for example, the Click Customers who want to visit the Gelato area. And then, submit the change through the ModifyBookingSchedule API. The BookingSchedule component will store this change in the database. At this time, our Observer Pattern will function, the CRM will find out this change and regenerate the Ticket for the corresponding Click Customer if it is needed. Cause it is not a part of the Management System, so not shown in this Figure.

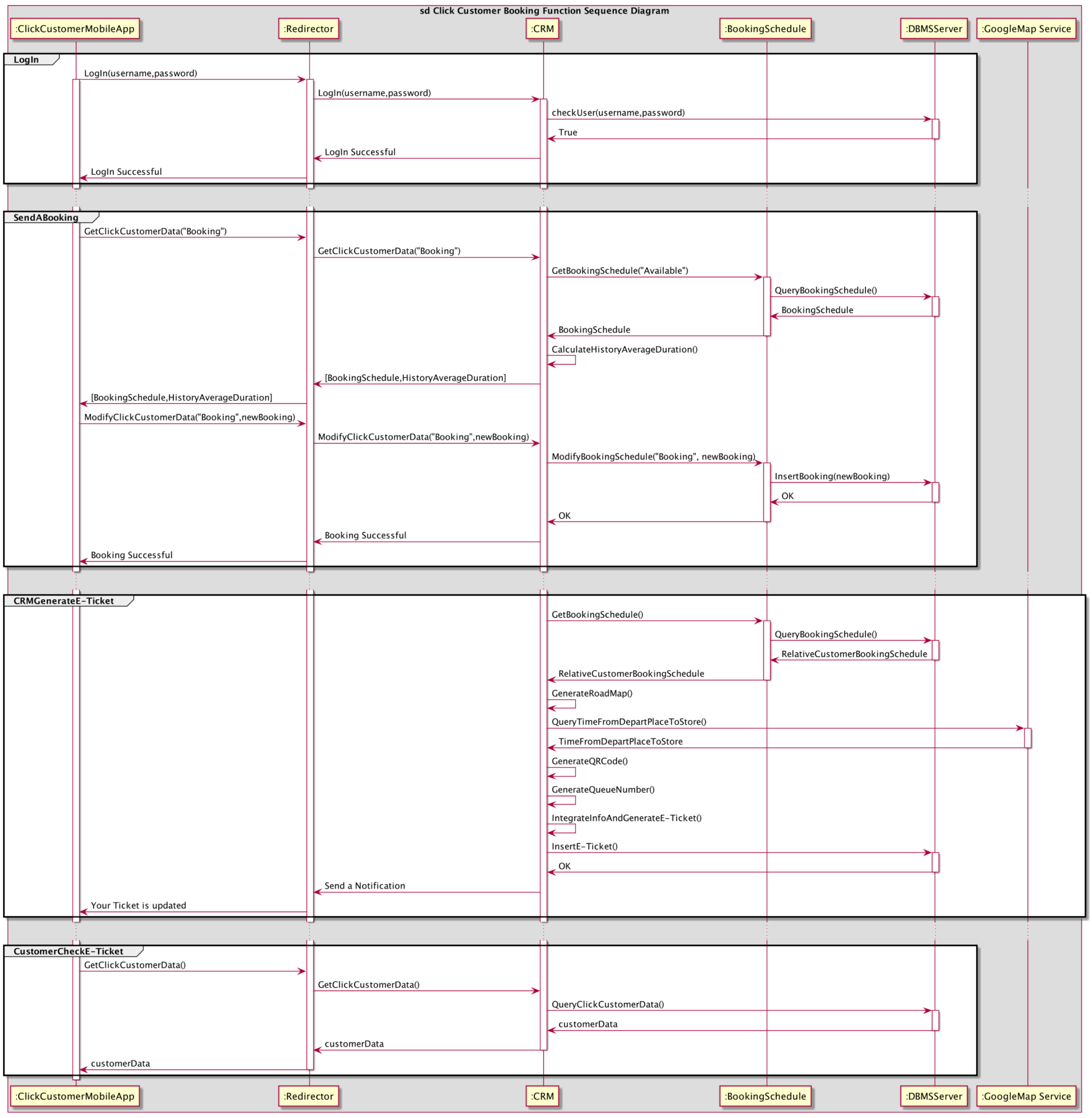


Figure 2.8: Click Customer's Booking Function Sequence Diagram

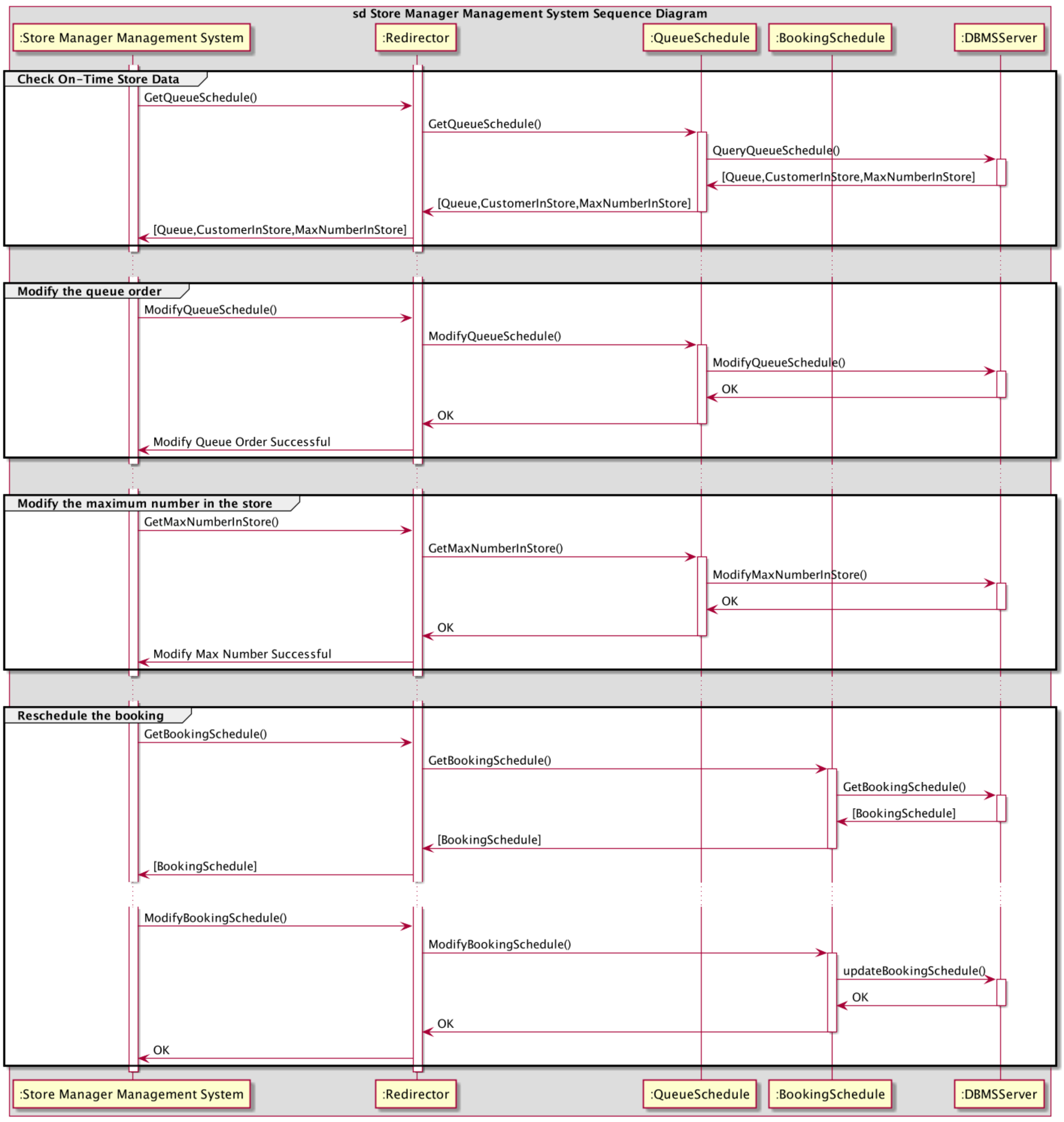


Figure 2.9: Store Manager Management System Sequence Diagram

2.5 Component Interfaces

The fundamental relationship between Interface and components has been illustrated in the component diagram Fig.2.3. So, in this section, we will introduce each Interface in detail.

For **CRM**'s Interface:

- RegisterNewClickCustomer(username, password): It is only for the Click Customer Mobile App to register the new User.
- LogIn(username,password): It is for registed user log-in.
- GetClickCustomerData(motive): It can get all Current/History Ticket. If the motive value is "Booking", it can also get the Historic Average duration and the available booking date/time for the Booking operation.
- ModifyClickCustomerData(): The Mobile App must use this function to contain the new Booking field and send the Booking. If the Booking is Successful, it can get the OK string, so that this Booking is already stored in the database, and the Click Customer just waits for the notification.
- SendNotifyToClickCustomer(): This function is used to receive notification from the server passively. For example, when the Ticket is generated or updated, the Mobile App will get the notification, and the Customer can check the new Ticket.

For **BookingSchedule**'s Interface:

- GetBookingSchedule(datetime, customer, item): It can use to get the wanted BookingSchedule. The range is datetime, customer, or item. All of these parameters are a list, a datetime range, customer list, or item list; this function will return the wanted Booking. For example, when the CRM component wants all the available dates/times in the schedule, it can give the current DateTime to this function. It will return all schedules from now, so the CRM can calculate and give the available date/time to the Mobile App. Alternatively, if the Store Manager Management System wants to get the Customer list who must visit Gelato and Pizza area, it can give the [Gelato, Pizza] to the item field. For sure, it will return all corresponding customers.

- ModifyBookingSchedule(): It used for the Management System, or CRM system want to insert or update some Booking, if the database accepted the change, it will return the OK string.

For **QueueSchedule**'s Interface:

- GetQueueSchedule(): It can return the Current Queue order, both the Queue and CustomerInStore. All our Presentation Layer's device will use it except the Mobile App and the QRCode Scanned Machine. The Digital Counterpart will use it to display the next queue number. The Ticket Machine will use it to display how many Customers in queue and store, the Management System will use it to check the Store In-Time Data.
- ModifyQueueSchedule(): Management System and Ticket Machine will use it to modify or insert the Queue order. Like the ModifyBookingSchedule, only if the database finally accepted this change, it will return the OK string.
- GetMaxNumberInStore(): the Ticket Machine will display the Allowed Maximum number in the Store to remind the Customer, and the Management System will also call it to display this value.
- ModifyMaxNumberInStore(): The Management System will use this to change the Allowed Maximum number in the Store value to reduce/raise its current customer number.

2.6 Selected Architectural Styles and Patterns

As the Sec.2.1 said, we adopted the **3-Tiered Architecture** style for our system, as Fig.2.10 in Sistem Information Book P.197 [1] showed. There are 13 kinds of 3-Tier architecture configuration. From Config 1 to 5 are Think Client and 6 to 13 are Thin Client, we decided to use the **Thin Client** like the Config 7. So that our Presentation Layer devices can applicate the lowest cost hardware, in this situation, we chose the Raspberry Pi 4 for all these devices.

Config. /Tier	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Tier 3	D	D	D	D	D,A	D	D	D,A	D	D	D,A	D,A	D,A, P
Tier 2	D	D	D,A	A	A	D,A	A	A	D,A, P	A,P	A,P	P	P
Tier 1	D,A, P	A,P	A,P	A,P	A,P	P	P	P	P	P	P	P	P

Figure 2.10: Classification Of 3-Tier Architecture

By the way, as the Sec.2.3 Deployment View mentioned, we chose the RACS and shared-disk strategy to handle our load balance problem. In that section, we also proposed two options to deploy our system and why we chose the second option.

Model-View-Controller - MVC: We referred to the Slide of the Software Engineering 1 course and decided to adopt the MVC pattern to implement our Presentation Layer devices. As the Fig.2.11 in Slide showed [4], the Controller is used to invoke the API and communicate with our Application Server. The View used to the UI, and the Model used to handle the data arrive from the Application Server and notify the View when something changed.

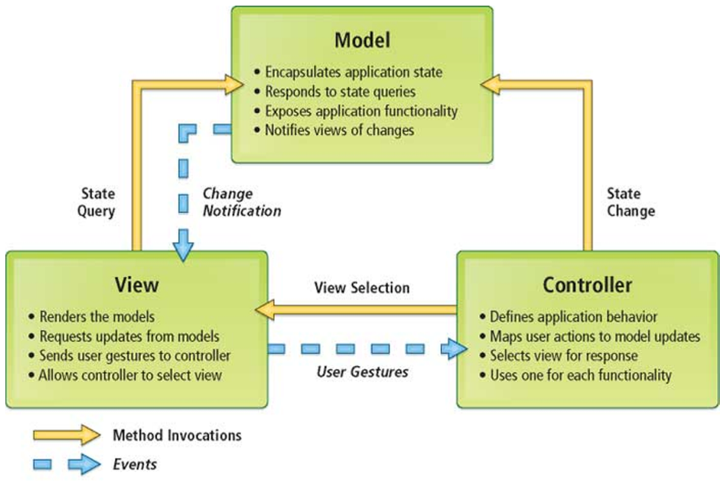

Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• Uses one for each functionality
Method Invocations
Events

Figure 2.11: MVC Pattern

Observer Pattern: We also referred to the observer pattern's idea like the Fig.2.12 below in the Slide [4]. Nevertheless, our system has a specific case: we treat the database as the subject and our CRM component as the Observer. While some Click Customer's Booking is changed, the CRM will send a notification to the Click Customer Mobile App. If it is necessary, the CRM will update the E-Ticket for the corresponding customer.

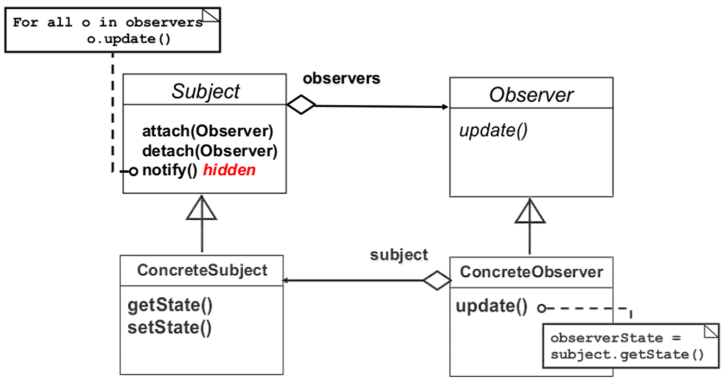


Figure 2.12: Observer Pattern

2.7 Other Design Decisions

In this section, we will talk about the Database Design decision. We referred to the Database 1 Course [6] and the Database System book [7].

Our data have several key features: Our data is structured, and the schema can be predefined, and our data structure is vertically scalable instead of horizontally scalable. So we decided to adopt the **Relational Database**.

In the database design process, we will strictly follow the design process. From Conceptual Design to Logical Design, and then to do the Normalization. Because the Relational Database is vertically scalable, if the initial design is wrong, the cost of later modification will increase exponentially.

For the deployment issues, as the Sec.2.3 Deployment View mentioned, we used **PostgreSQL** as the Database Management System because it is free, open-source, mature, and stable.

Chapter 3

USER INTERFACE DESIGN

3.1 UI design

We illustrated some main UI designs in RASD document, here, we describe more details for the specific UI button.

UI-Date Time After enter the booking button, there is an option of date and time, Fig.**??**, the CLup will give the available booking date, which with the deep dark color. And the same method for selecting available booking time.

UI-Duration Time It needs the user to estimate his shooping time, Fig.**??**, for long-term users, they can press the "Analysis" button, the system will approximate a duration time depending on his shopping history.

UI-Ticket Fig.**??** shows the users current ticket and history ticket. The current ticket with the red point, to inform the user it is the valid ticket.

UI-Ticket PDF it present the PDF ticket for the user, Fig.**??** the QR-code using for scan on the entry. And it also give the suggestion of departing time from current position to the store.

UI-Notification Fig.**??** which display the notification of rescheduled ticket, the red point means the unread message. And it will also give the new ticket to the user.

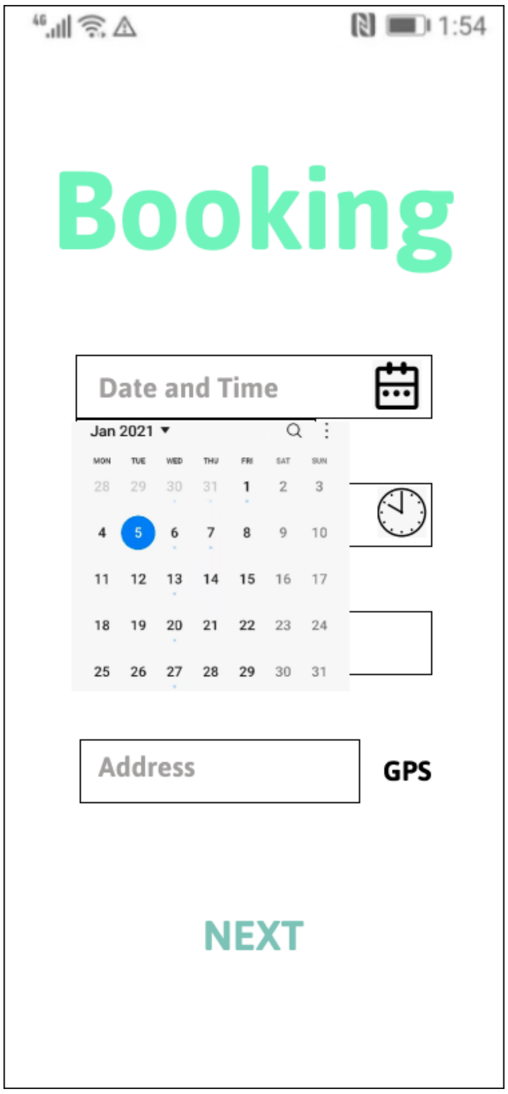


Figure 3.1: UI-Date

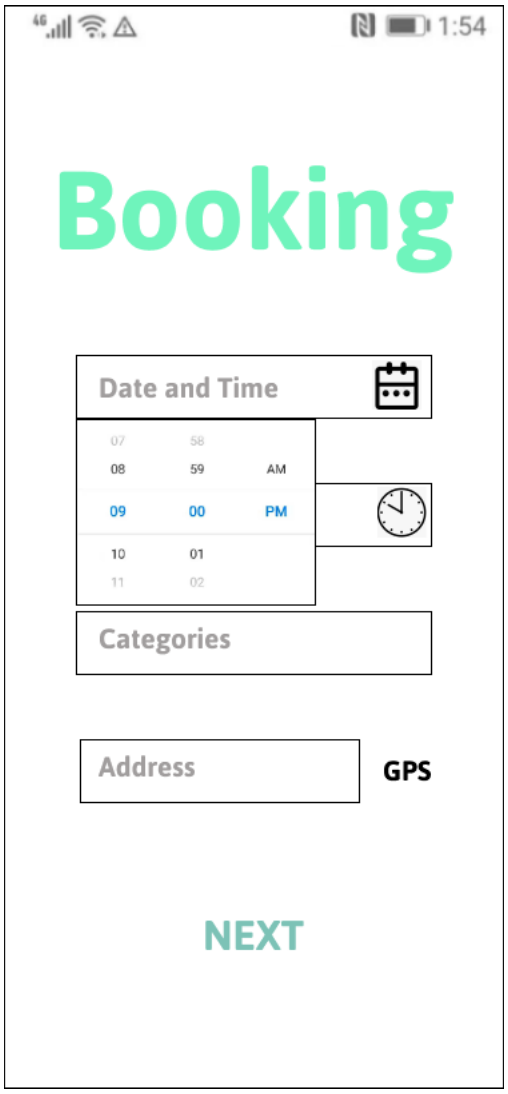


Figure 3.2: UI-Time

3.2 UX design

We present the UX diagram of user, Fig.3.3 the diagram show the proceed of how the user from login to get the valid ticket. Corresponding to the UI design, it show the total user view of the CLup application.

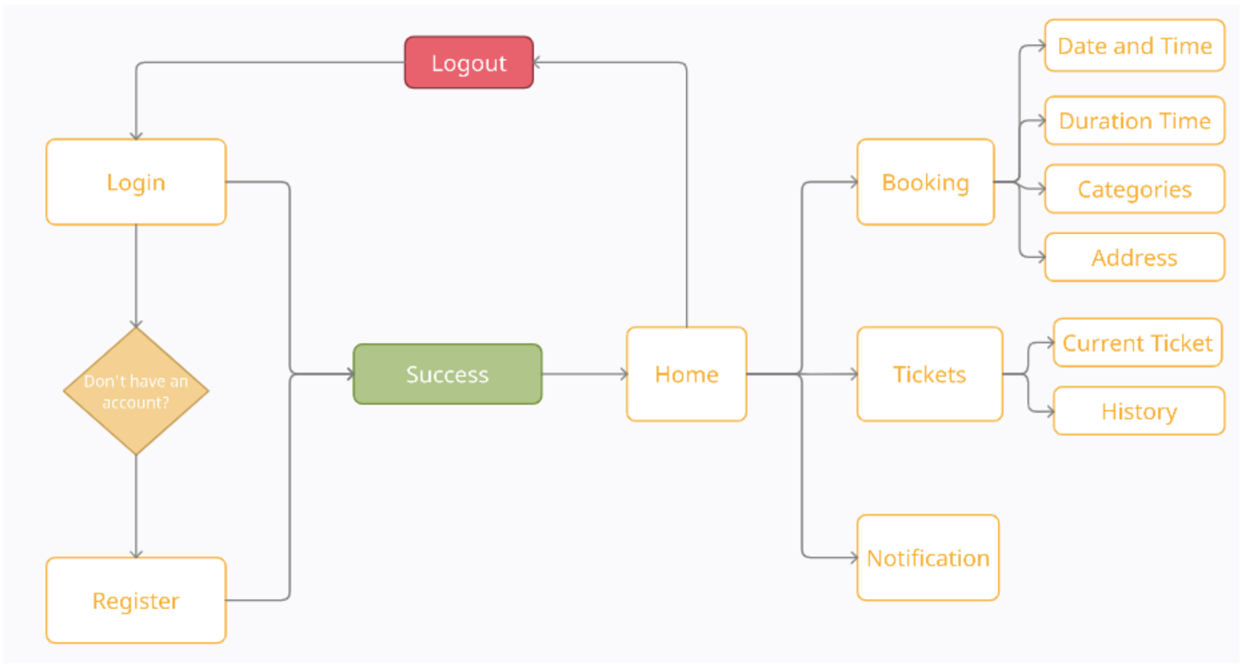


Figure 3.3: UX-User

Chapter 4

REQUIREMENTS TRACEABILITY

Chapter 5

IMPLEMENTATION, INTEGRATION AND TEST PLAN

The system will be implemented, intergrated and tested. For these part are very important and necessary for CLup application. We need to complete it as much as possiable. We can mainly divided into three subsystems:
(1) the user mobile
(2) the manager web server
(3) the application server

5.1 Implementation

We want the implementation of CLup application is efficient.Here are the order of system to be implemented:

(1) The user application will be implemented
(2) The database implemented to collect and save the data.
(3) Now the manager serve will monitor the system and control it

After we will specific the order of modules that the developer need to develop:

• Booking manager
• Data storage manager
• Reschedule manager

- Queue order manager
- Duration time estimate manager

The request application server will implement the following order:

- User application manager
- Specific information request manager
- Data storage manager
- shopping route suggestion manager
- Depart time suggestion manager

On the User application, there will be the following modules, it will be implemented in this order:

- Booking manager
- request manager
- Data manager

5.2 Integration

The aiming od integration testing is at exercising: interface and modules interactions. We need to consider and avoid the different integration faults:

- Side effects on parameters or resources
- Violations of value domains, capacity, or size limits
- Omitted or misunderstood functionality
- Nonfunctional properties
- Dynamic mismatches[5]

Our integration testing based on the Top-down structure of system with the thread of a portion of several modules offers a user- visible function.

5.3 Test Plan

We will use the black-box testing after the integration. We will use modules to devise test cases, and the actural behavior of software under test is checked against the the behavior specified by the model. The prerequisties of performances testing is expected workload and acceptable performance. For system testing, the teams are independent and will test the functional and non-functional requirements. As for the test environment, it need to be as close as possiable to production enviroment. In performance, we need to identify the inefficient algorithms, optimize the query possibilities and hardware/ network issues.

Chapter 6

Effort Spent

- **Kong XiangYi**

Date	Task	Hours
2021/01/02	Group discussion and task assignment	2h
2021/01/04	UI design	2h
2021/01/05	UX design diagram in chapter 3	2h
2021/01/06	Complete UI and UX design in chapter 3	4h
2021/01/07 2021/01/05	Complete the chapter 5 implementation, integrate and test UX design diagram in chapter 3	4h 2h
2021/01/08	Group Discussion	2h

- **Zhang YueDong**

Date	Task	Hours
2021/01/01	Launch DD	2h
2021/01/02	Group discussion and task assignment	2h
2021/01/03	Did S.2.2.1 and S.2.2.2 Component Diagram	3h
2021/01/04	Did S.2.2. Component Diagram describe.	2h
2021/01/05	Did S.2.3. Deployment Diagram and describe.	5h
2021/01/06	Did S.2.4. Runtime View and describe	4h
2021/01/07	Did S.2.5. Component Interface describe	2h
2021/01/08	Finished the architecture part and did the group discussion.	4h

Bibliography

[1] Cinzia Cappiello, Mariagrazia Fugini, Paul Grefen, Barbara Pernici, Pierluigi Plebani, Monica Vitali. (7 settembre 2018) **Fondamenti di Sistemi informativi**: per il Settore dell'Informazione.

[2] "IEEE Standard for Information Technology–Systems Design–Software Design Descriptions," in IEEE STD 1016-2009 , vol., no., pp.1-35, 20 July 2009, doi: 10.1109/IEEESTD.2009.5167255.

[3] "ISO/IEC/IEEE Systems and software engineering – Architecture description," in ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000) , vol., no., pp.1-46, 1 Dec. 2011, doi: 10.1109/IEEESTD.2011.6129467.

[4] Gianpaolo Cugola, Lorenzo Di Tucci. (A.Y.2018/2019) The slides of the Software Engineering 1 course. Politecnico di Milano.

[5] Elisabetta Di Nitto, Matteo Rossi. (A.Y.2020/2021) The slides of the Software Engineering 2 course. Politecnico di Milano.

[6] Sara Comai, Giovanni Meroni, Davide Piantella. (A.Y.2019/2020) The Database 1 course. Politecnico di Milano.

[7] Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Riccardo Torlone. **Database Systems** concepts, languages & architectures. The McGraw-Hill Companies. ISBN 007-709500-6.