POLITECNICO
DI MILANO

Politecnico di Milano
A.A. 2015-2016
Software Engineering 2
"myTaxiService"

Design Document (DD)

version 1.0

Annalisa Rotondaro (mat. 854268)

Lorenzo Federico Porro (mat. 859093)

Release date: 04 dicembre 2015

Contents

1 Introduction

1.1 Purpose

This document contains all the design choices and the components that the system need to satisfy functional requirements specified in section 1.3.2 of the RASD document. The design document referes to this audience:

- Developers;
- Stakeholders:
- Other software designers that collaborate to the design choices;

- Anyone in charge of the analysis and validation;

The goal of this document is to describe the design of the application and to show components, their interactions and the main choices about design, architectural styles and management of the non functional requirement like availability ("The system should be available 24 hour a day 7 days a week").

Inside the document will also be provided diagrams that show the way in which the components of the application interact and how the system provide the functional and some non functional requirement described in the RASD and the tasks that have to be provided.

1.2 Scope

The aim of the project is to optimize the taxi service of a city, in particular to simplify the access of passengers to the service, let taxi drivers work in an optimized way ensuring a fair management of taxi queues. So the scope is to create a system that have both a web app "myTaxiService.com" and a mobile app "myTaxiServiceApp" that satisfy this scope.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- **System:** the main system where the logic of the application is hosted;
- **Web app:** the site accesible by any browser;
- **Mobile app:** the application from which a user or a driver is able to access myTaxiService services;

1.3.2 Acronyms

- **RASD**: Requirements Analysis and Specification Document.

1.3.3 Abbreviations

- **DB:** DataBase;

1.4 Reference documents

- **Requirements Analysis and Specification Document** (RASD).
- **Assignament of the project.**
- **"Template for the design document".**

1.5 Document structure

The document is composed by 6 sections that deals with the choice on design level about MyTaxiService application:

- **Section 1 "Introduction"**: contains the purpose, the audience of the document and the scope of the application. It contains also all the definitions that are useful to read the document, some acronyms, some abbreviations that will be used throughout the document and the reference of the documents, that shows on what the document is based.

- **Section 2 "Architectural design"**: This section describes the architectural design choosen for the application and the motivation and advantages of this choice. We will describe the main components of the system in the subsection 2.2, and the interactions that they have each other and with the people that uses the application . The section 2.3 shows the architectural style choosen for this application and describes the motivation of this choice taking into account all the work done until now and the RASD document related to this document. We also show diagrams in subsections 2.4 and 2.5 where the reader can learn more details of the interactions of the components and how the components are mapped on the hardware of the application. The subsection 2.6 describes the behaviour of the system for the various functionalities that have to be provided and more detailed diagrams in which the main tasks are explained. The "components interfaces" subsection shows a more detailed picture, with respect to the one present in section 2.1 ("product prospective") of the RASD document, in which we describe the various interfaces that the components of the application have and what the interfaces of the components require and what they offer to work in a proper way. The subsection 2.8 describe other design decisions that have been taken to complete all the choices regarding the design.

- **Section 3 "Algorithm design"**: This section describe a more detailed concept and go toward a detailed design step. This part is not present in the commons design document but here there is a high level algorithm of one main functionality of the system, advised by the authors of the document but without putting any kind of limits to the software developer, which is free to choose whether use the proposed algorithm or not. This part is made to just give a hint for an high level algorithm.

- **Section 4 "User interface design"**: Here are the main user intefaces that the client and taxi driver will face during the usage of the application. This section will refer to the section 3.1.1 of the RASD document where there are some mockups of both the user interfaces of the web site and of the user interfaces of the mobile application.

- **Section 5 "Requirements traceability"**: This section contains tables that show how the requirements present in the RASD document are

mapped onto the design elements (components). For this purpose there is a table in which you can see how functional requirements are mapped onto the components defined in section 2.2.

- **Section 6 "Supporting informations"**: In this section are present, first of all, all the informations and definitions that are needed for the reader to understand all the concepts in this document (see glossary section 6.1). Then all the bibliography, documents and other resources that have been used to redact this document are present in section 6.2 "references". The subsection 6.3 contains all the tools used to redact this document, including tools that helped to write all the diagrams and all the pictures . The last subsection 6.4 contains the number of hours for each author spent to redact the document.

2 Architectural design

2.1 Overview

The myTaxiService application expect that if a client would like to call a taxi, the request is sent to the system that is in charge of checking if there is a taxi available for that request and assign this taxi to the client. The same for a reservation, so if a client wants to reserve a taxi, he has to use the application and insert all the required data for the booking, for example the date and the hour of the appointment. The system that receive this requests has to check if the hour respect some bounds (in this case if is at least two hour before the pick up time) and have to guarantee that the taxi that will pick up the client at the appointend time, will work for that client and not for the others.

Regarding the taxi drivers, if, for example, one of them would like to work, he have to click the button "i'm available" in the user interface of the mobile application and the system has to insert that taxi in the respective queue according to the GPS coordinates. In figure 1 there is a high level schema based on the description given before in which you can see the actions made by taxi drives (men in green) and those made by the clients (men in blu) made by their devices to a "big" system that it will take care of requests and will respond according to circumstances.

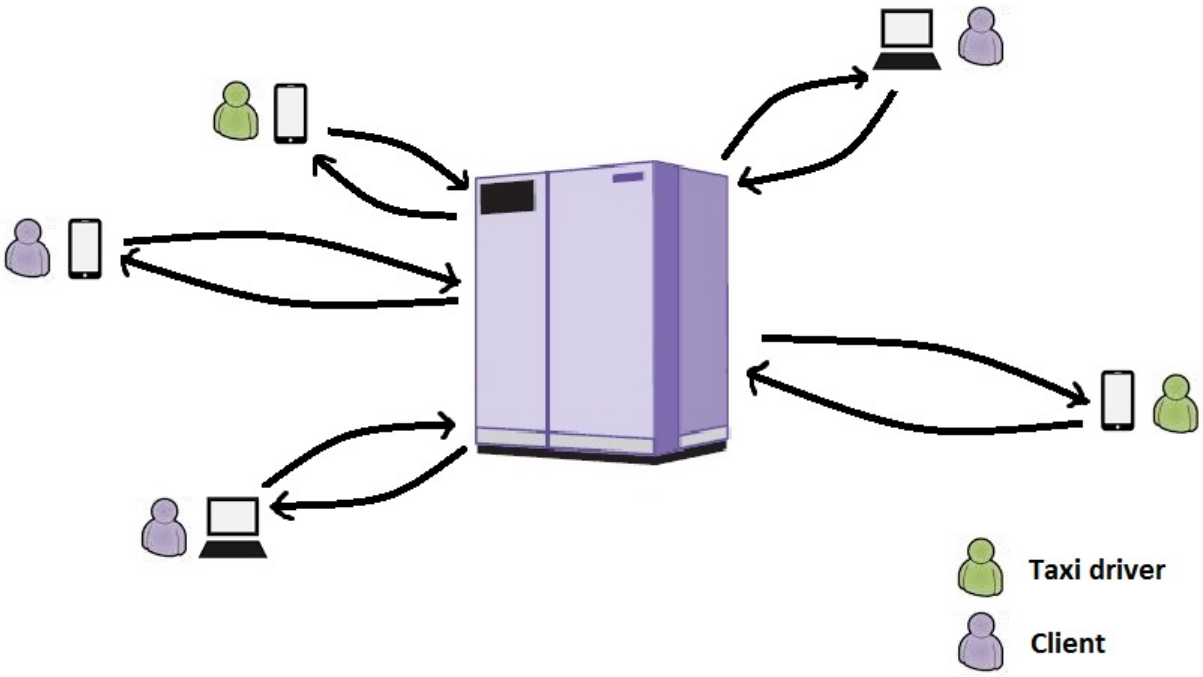

Figure 1: An high level of the system architecture

The following sections contains components and architecture style in general that have been chosen for the application.

2.2 High level components and their interaction

For this application we decide to have some components that each is in charge of one functionality. In the component view subsection and runtime view we will see the possible interactions among them with some schemes.

2.3 Selected architectural styles and patterns

After the description of the application done in the section 2.1, we decided to choose client-server architectural style because, given the structure of the application, there is one system, the server of the application, that has to manage:

- Requests made by all clients that would like to call or reserve a taxi;
- Responses of taxi drivers that would accept the call;
- Requests by taxi drivers to make them availaible or not.

In particular we decided to have a thin client (see the glossary in section 6.1), given the fact that the only things that the client has to do are, for example, insert some data to sign in and press a button to call a taxi. Instead the server will handle all the data management (it have in the database all the data), as well as the processing of requests.

We also choose to separate user interfaces depending on the type of user logged in. In fact, as noted in section 3.1.1 of the RASD document, there are two different personal pages depending on which type of user is logged in. This because there are distinct functionalities on the basis that you are client, (he can call or reserve a taxi), or taxi driver (he can become available, become unavailable or response to a call). All the choices are showed in the design space in figure 2.

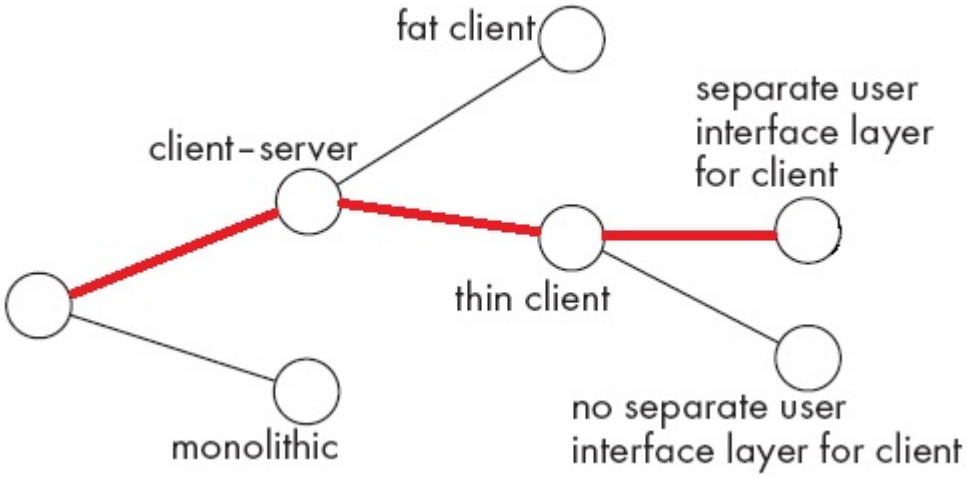

Figure 2: Design space of the application

For the design pattern we choose a three-tier (figure 3), this because, given our application myTaxiService, there is a clear division of the application in three different modules or layers respectively dedicated to user interface (which interfaces with the client and that will be mapped, as physical layer, on the devices of customers and taxi drivers), the functional logic (business logic, which manages everything related to the processing of the application) and management of persistent data (data on the server side in the database).

These modules are designed to interact with each other along the general lines of the client-server paradigm (the interface is the customers of the business logic, and the latter is the customers of the modules of management of persistent data) and using well-defined interfaces. In this way, each of the three modules can be changed or replaced independently of the others giving scalability and maintainability to the application, which does not happen , at least at this degree, if you take into account the lower tier as tier 1 or 2 .

So the decoupling of logic and data, logic and presentation makes the whole system more maintainable because if happen to have to change some things in one of the three-tier, (for example presentation), the change does not affect the others (middle tier and storage tier), so I have to change only the tier that need some changes.

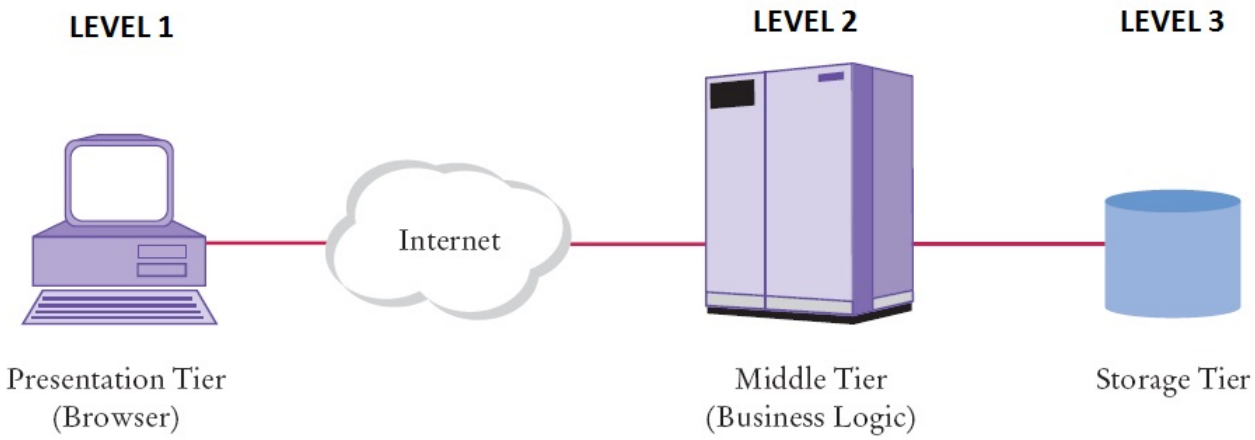

Figure 3: Pattern three tier schema

In fig 4 is showed a practical example of some performances of the various type of design patterns, from 1 to 4 tier. Here we can observes the differences of the uptime and downtime. We choose the three-tier that have 1.6 hours of downtime in a year, a time so small that can be neglected and therefore we can consider the system running 24 hours on 24, 7 days out of 7. Time of downtime of tier 1 and 2 are respectively 28.8 hours and 22 hours in a year , always small but more than the three-tier. Regarding the uptime , we have 4 nines for the four-tier, three nines for the three tier while only 2 nines for the lower. We also have the presence of redundant components for the system , to ensure a high uptime , and then the availability property. In the myTaxiService application in fact, we expect the presence of a second copy of the component " Data " (another database that kicks in when the main go down) and a second copy of the component " Application " .

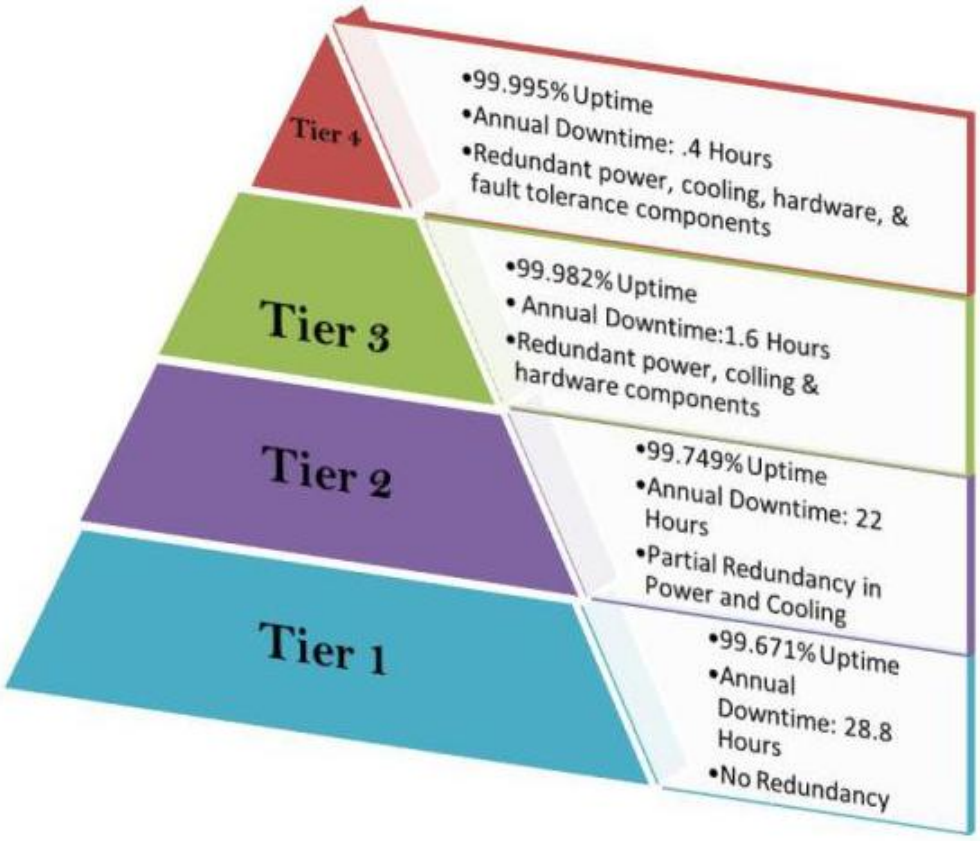

Figure 4: Practical example of performances about the four type of design patterns.

At first glance, the pattern model-view-controller (see the glossary in section 6.1) looks similar to the three-tier pattern that we choose. In fact, the two patterns have both the presentation part ("Presentation tier" in one and "View" in the other), the logic part ("Business logic" in one and "Controller" in the other) and the data part ("Storage tier" in one and "Model" in the other) but what changes, which is why we choose the three-tier instead of the model-view-controller is their interaction.

In the three-tier pattern the user interact with the presentation tier using an user interface, then the business logic manages the actions received from the presentation layer and uses the data in the storage tier to do so. We came to the conclusion that the elaboration of the level 2 communicates with the client through its user interface of the level 1. The pattern is very linear and is just what we need for our application . For example, if a client would call a taxi, he push the button call in the user interface, the request go to level 2, which will question all taxi drivers needed in the queue (information that is in

the storage tier , then in the database). Found the taxi driver available and changed his state (change in the database , from free to busy) the level two announced the result at the client through the user interface in the level 1. So all the comunication must go through the middle tier.

As regards the MVC instead (see figure 6) , the user invokes the controller (the logic of the application) and then the controller updates the model (the data part) . The model then updates the view which will show the change to the user. Is therefore a triangular structure that is not indicated in this application because we do not want that the database notifies the view of the client: this because We don't want to show all the informations of the changes of the database , such as when the driver refuses to a call and goes to the end of the queue .

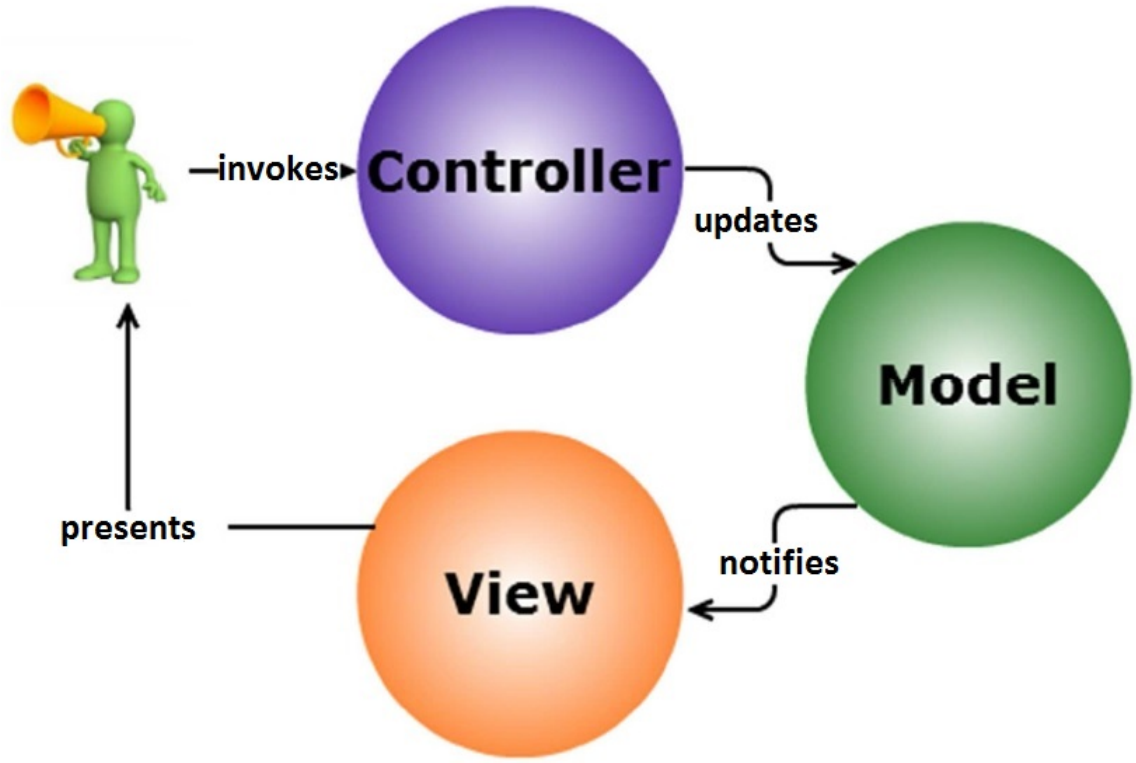

Figure 5: Pattern MVC schema

2.4 Component view

In this section will be presented diagrams for the main functions of the system. Each diagram refers only to the function specified, not to other related functions.

High level view

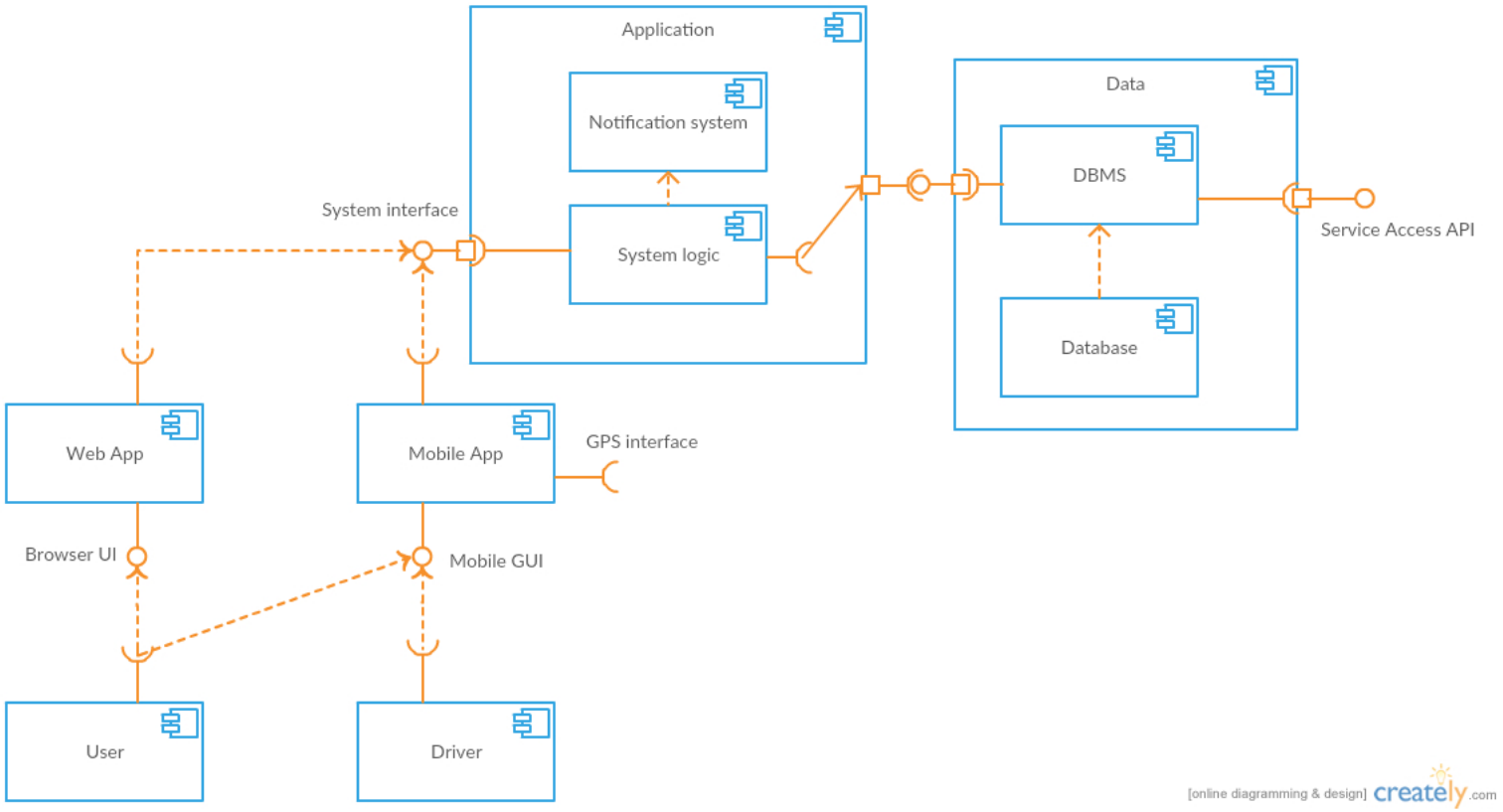

Figure 6: High level component view

The data component handles all the data and also regulates the access to them. The application contains all the code needed to elaborate an output given the inputs from the user's interfaces and the data stored inthe database. Both Mobile and Web applications handle the users' inputs and take care of showing to them all the relevant infos that the application component produces.

Login and registration view

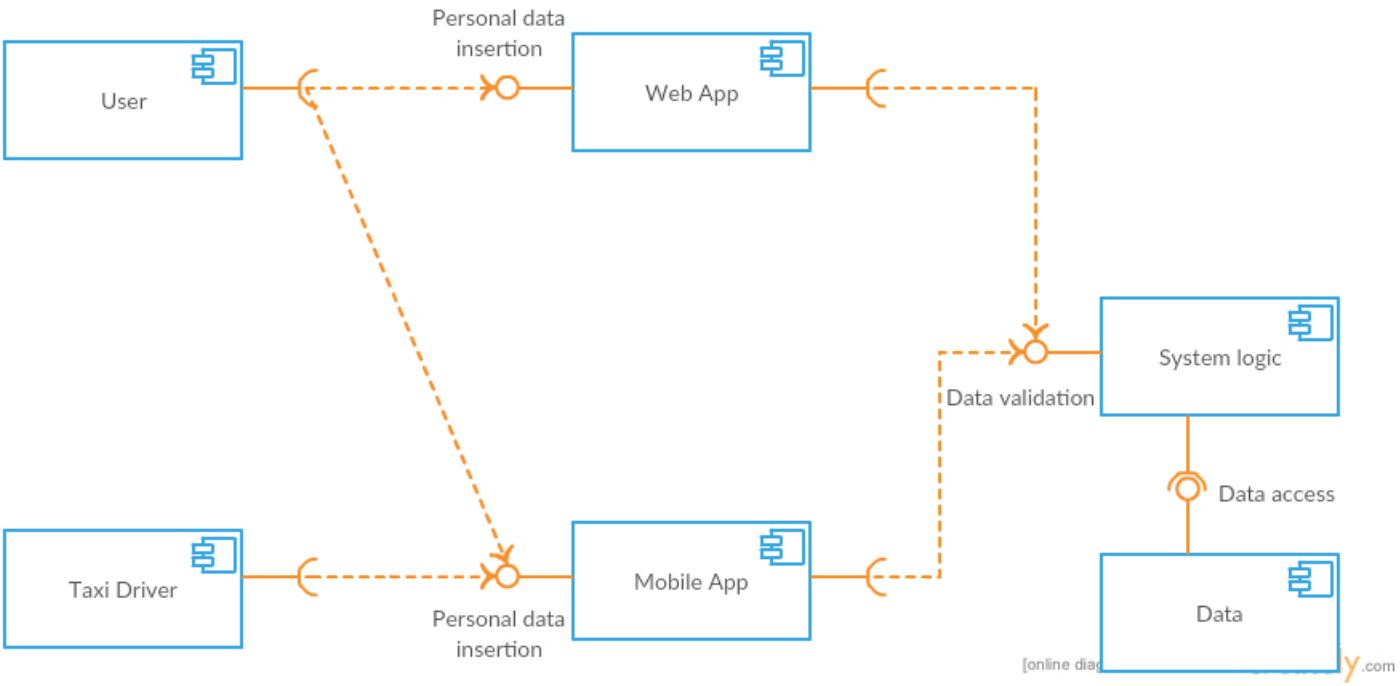

Figure 7: Component view for login and registration operations

Users and Drivers inserts their personal informations inside the forms given by both the apps. Then the system logic checks the database to validates the inputs and eventually register them into the database.

Call view

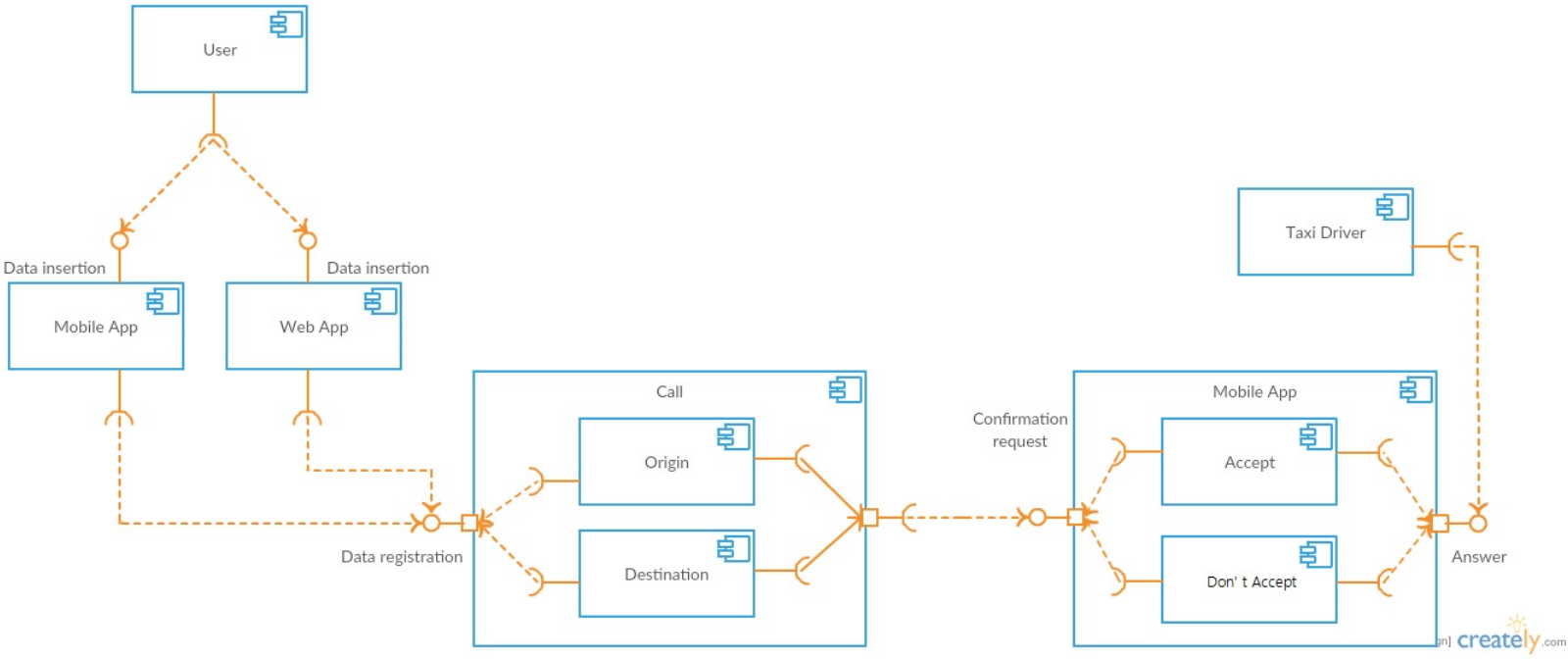

Figure 8: Component view for standard call operation

Call component is inside the application main component. The users make requests form any of the apps, then the system register the call and through the mobile app checks for drivers availability.

Reservation view

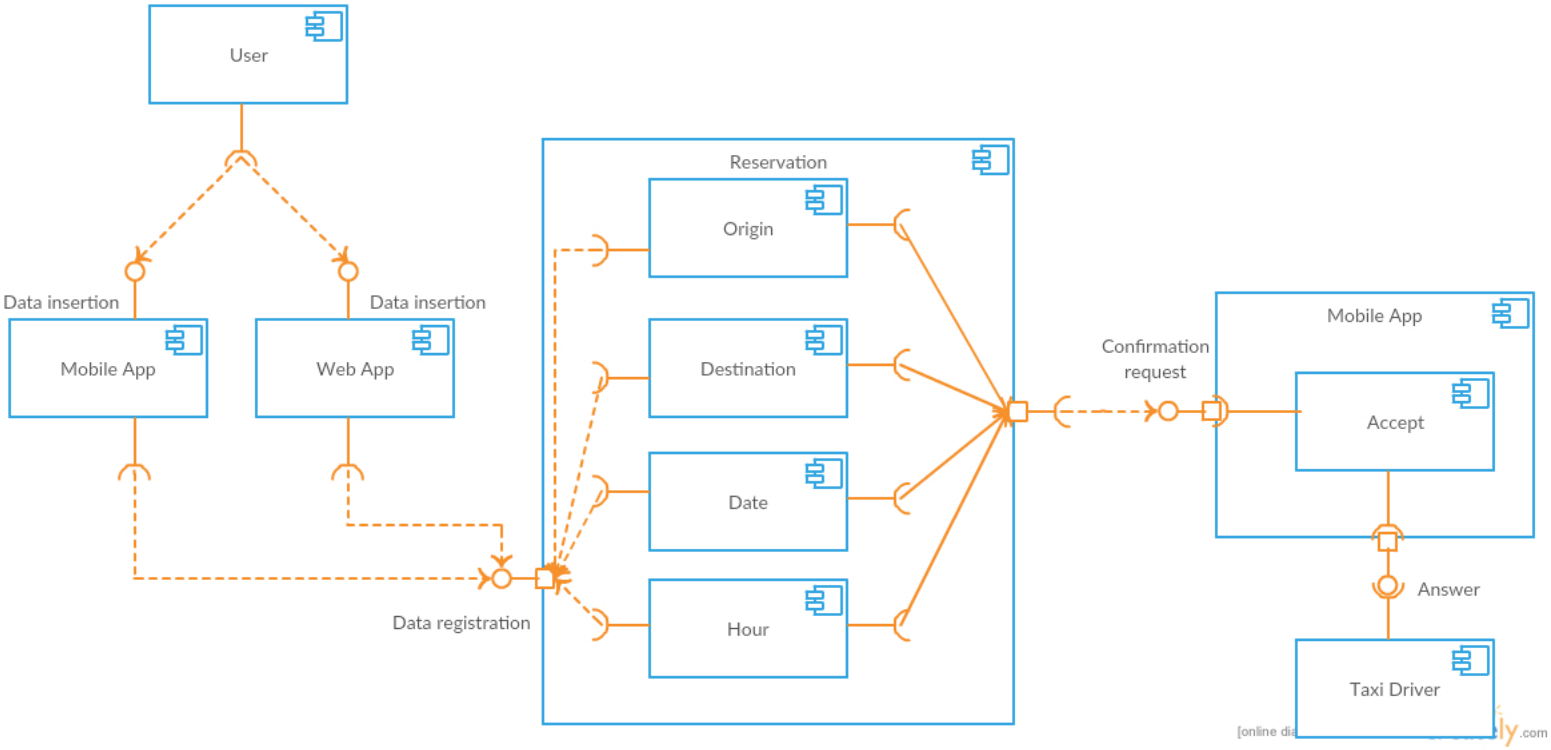

Figure 9: Component view for reservation operation

Reservation component is inside the application main component. Agents involved are the same as the call view with the diffrence that the driver assigned to the reservation can' t refuse the pickup.

Queue management view

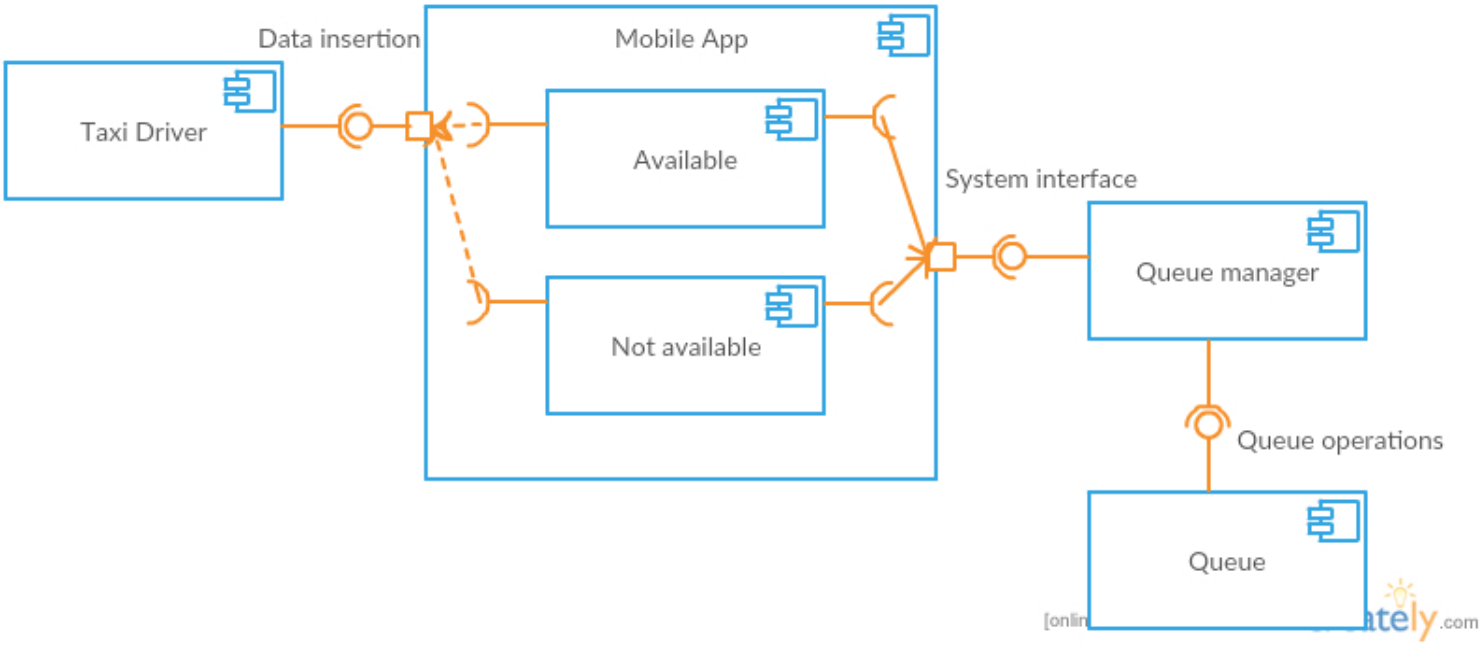

Figure 10: Component view for queue management operations

All queue components are inside the applciation main component. The queue is modified both by direct input of drivers (when they set their availability status) or directly from inside the queue manager when calls are arranged and drivers need to be removed from the queue.

Notification view

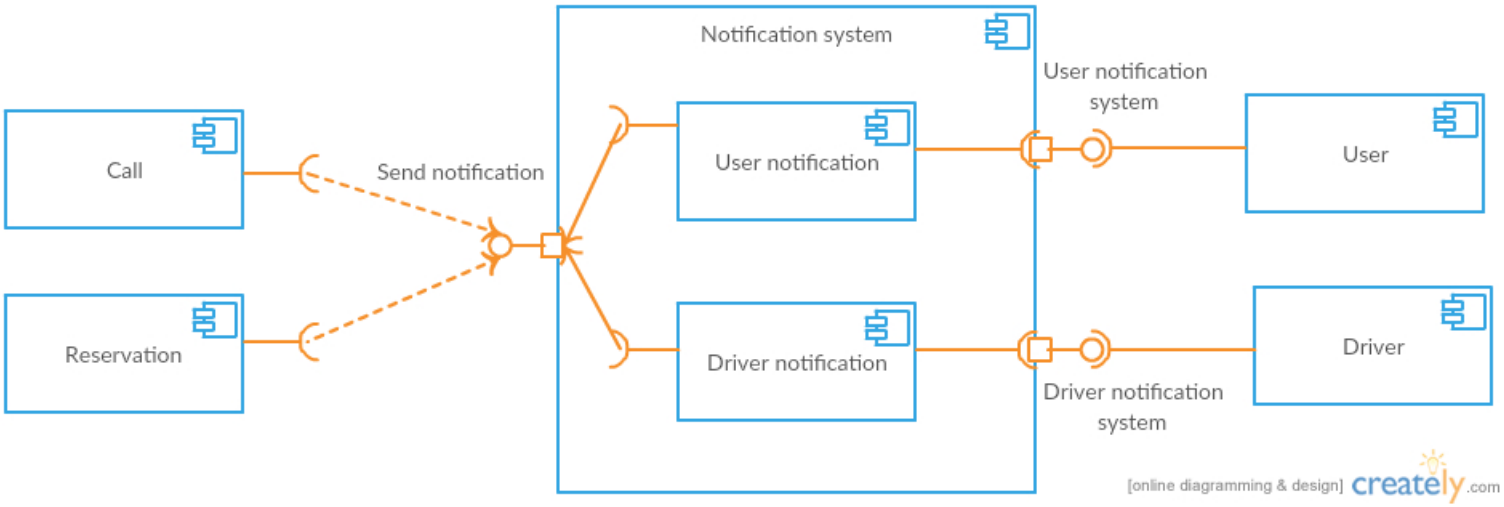

Figure 11: Notification system component view

The applicationmain component shows messages to clients through the notification system. In this diagram Web and Mobile apps are not shown to avoid redundancy as their only purpose is to display notifications to clients, the apps are implicit inside User and Driver notification systems interfaces in the diagram.

Data management view

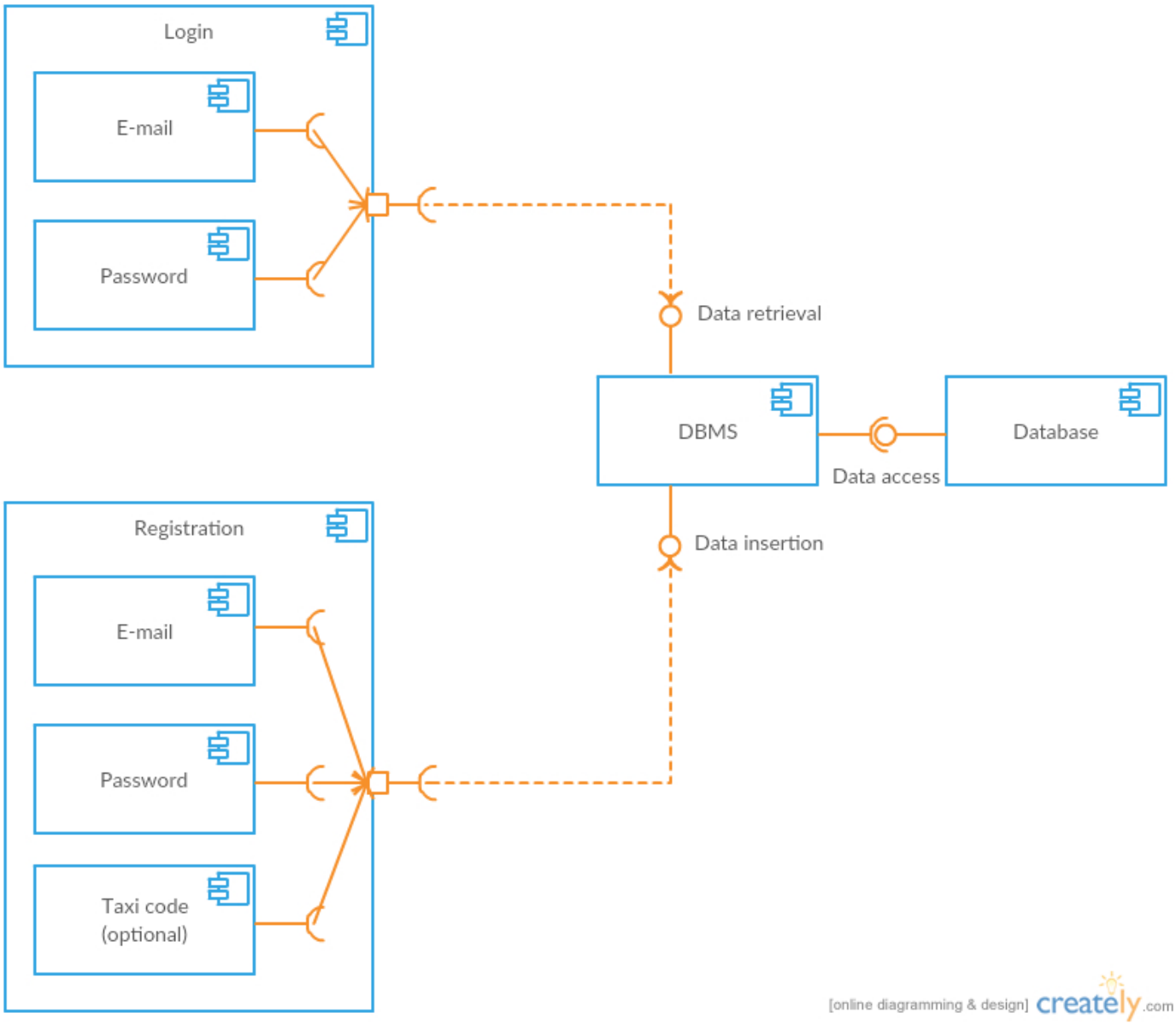

Figure 12: Data management system component view

All the personal data that users insert into the system are taken by the Application component and registered into the database through the DBMS. To

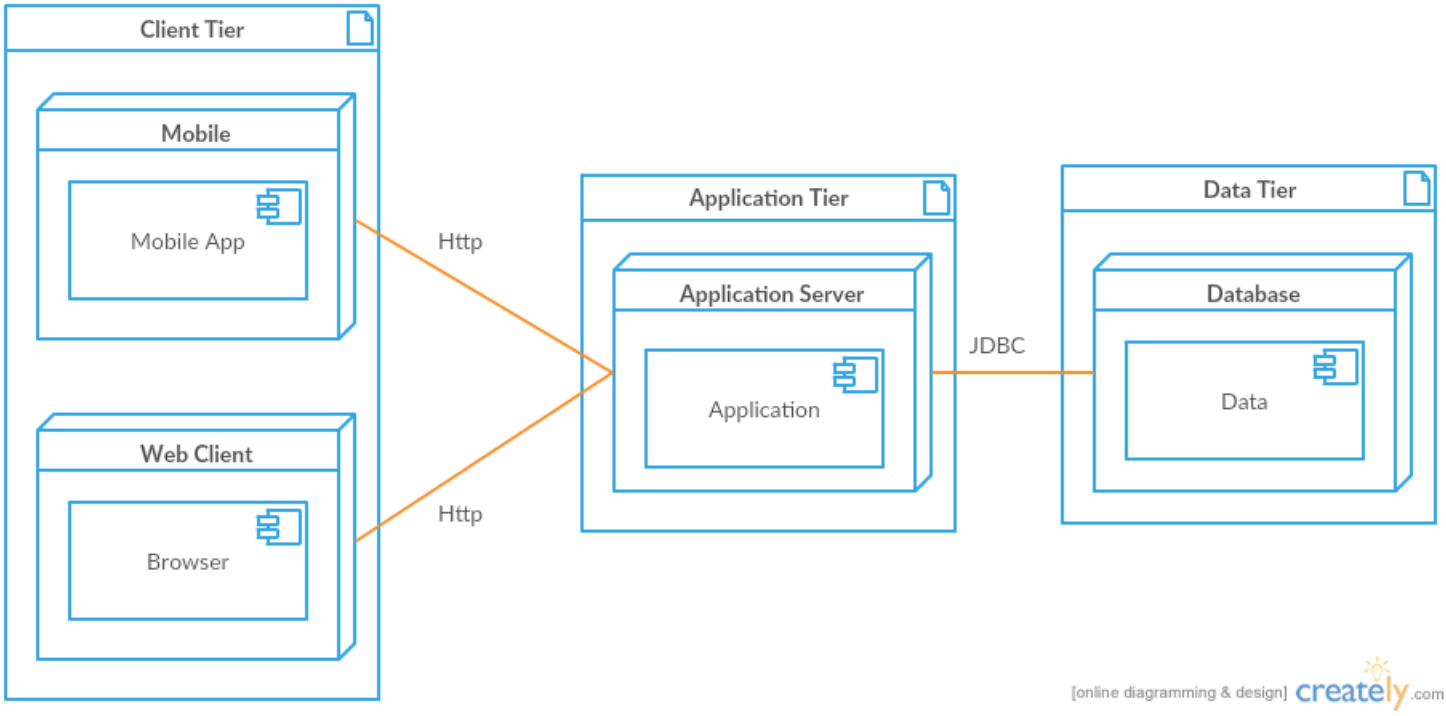

Figure 13: Deployment diagram for the whole system

validate a login data are retrieved in the same manner.

2.5 Deployment view

The following diagram maps software distribution on the different hardware devices.

The system uses a three tier architecture and, as such, each tier is on a different machine. The data tier is hosted on a NAS, all the data are copied on an identical backup machine to achieve system stability and recovery. The application is on a server machine and also here a secondary machine is present to guarantee 24/7 uptime while maintenance is in progress. Client tier is hosted either on mobile devices or on browsers.

2.6 Runtime view

The following diagrams display the runtime behaviour of the system during each operation. Each diagram refers exclusively to he operation specified. Diagrams are self explanatory.

Registration operation

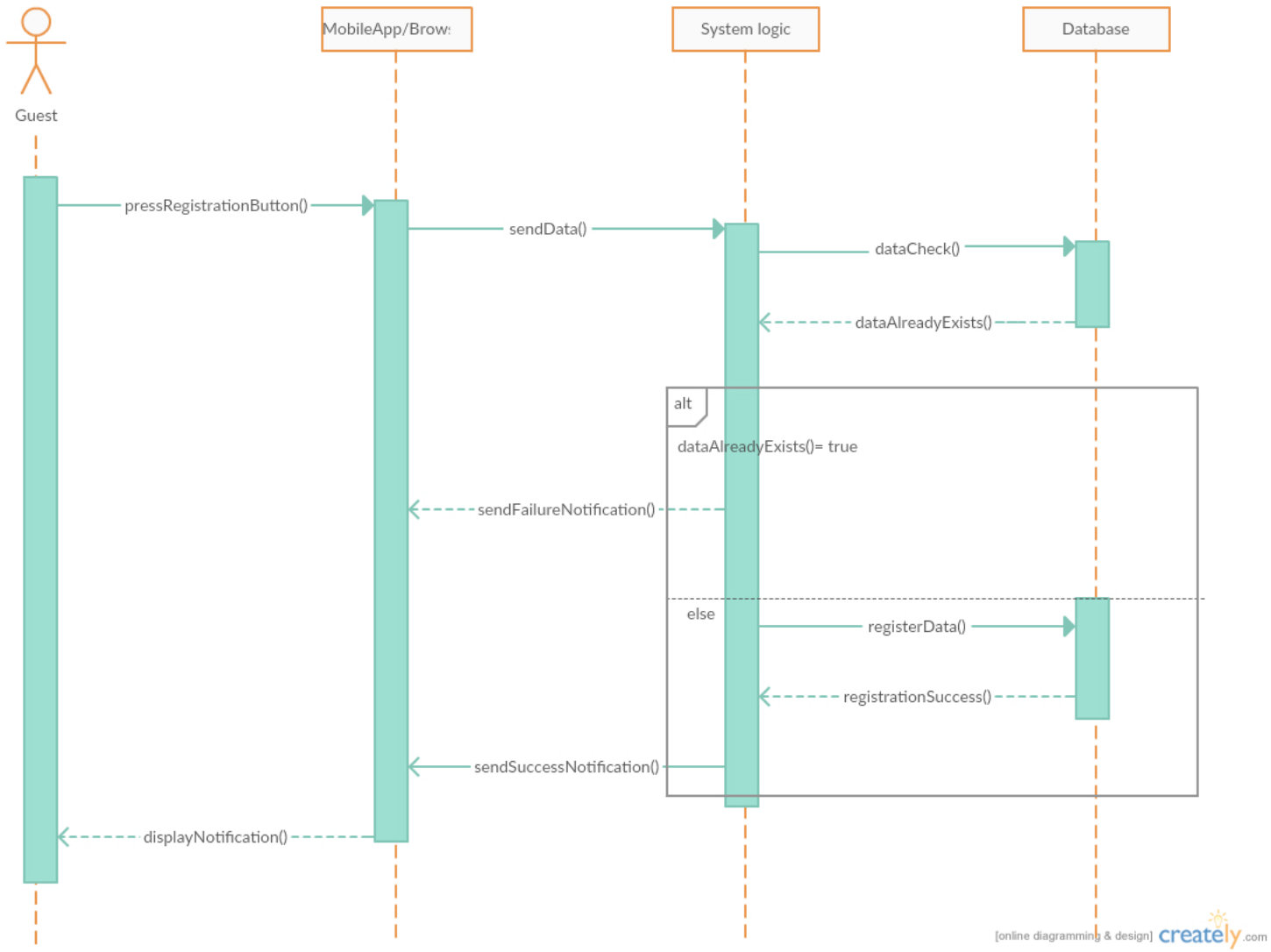

Figure 14: Sequence diagram for registration operation

Login operation

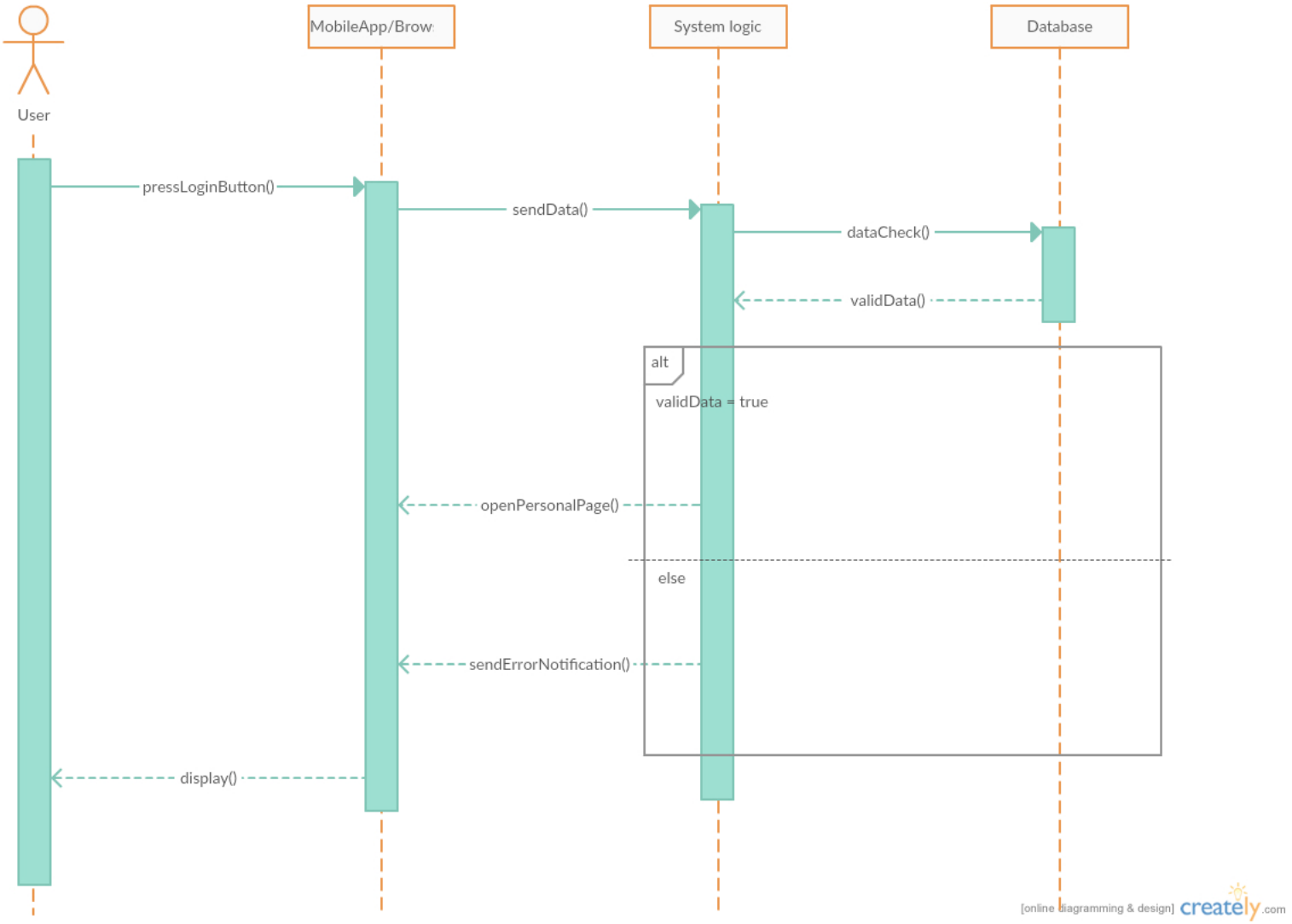

Figure 15: Sequence diagram for login operation

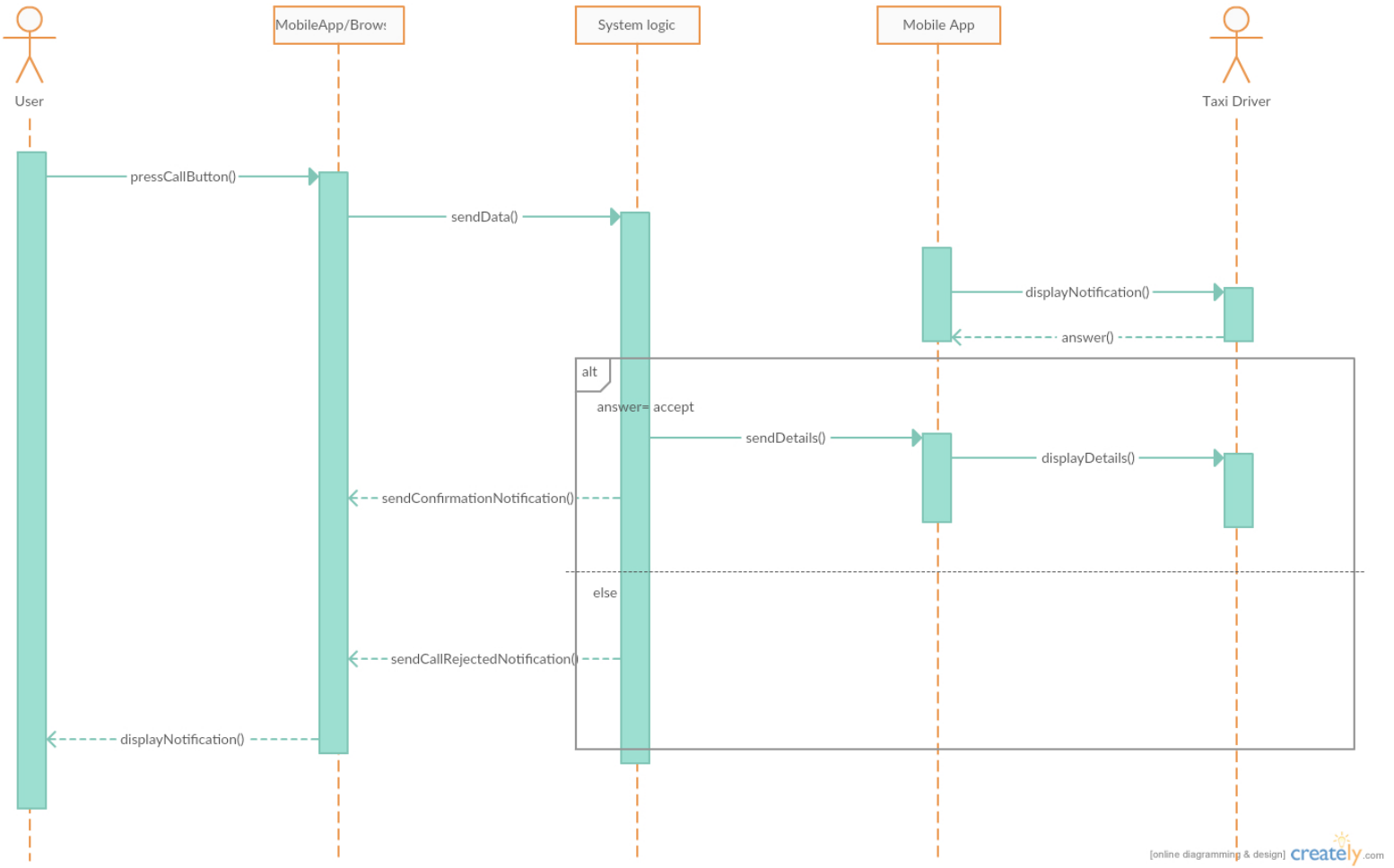

Figure 16: Sequence diagram for standard call operation

Call operation

Reservation operation

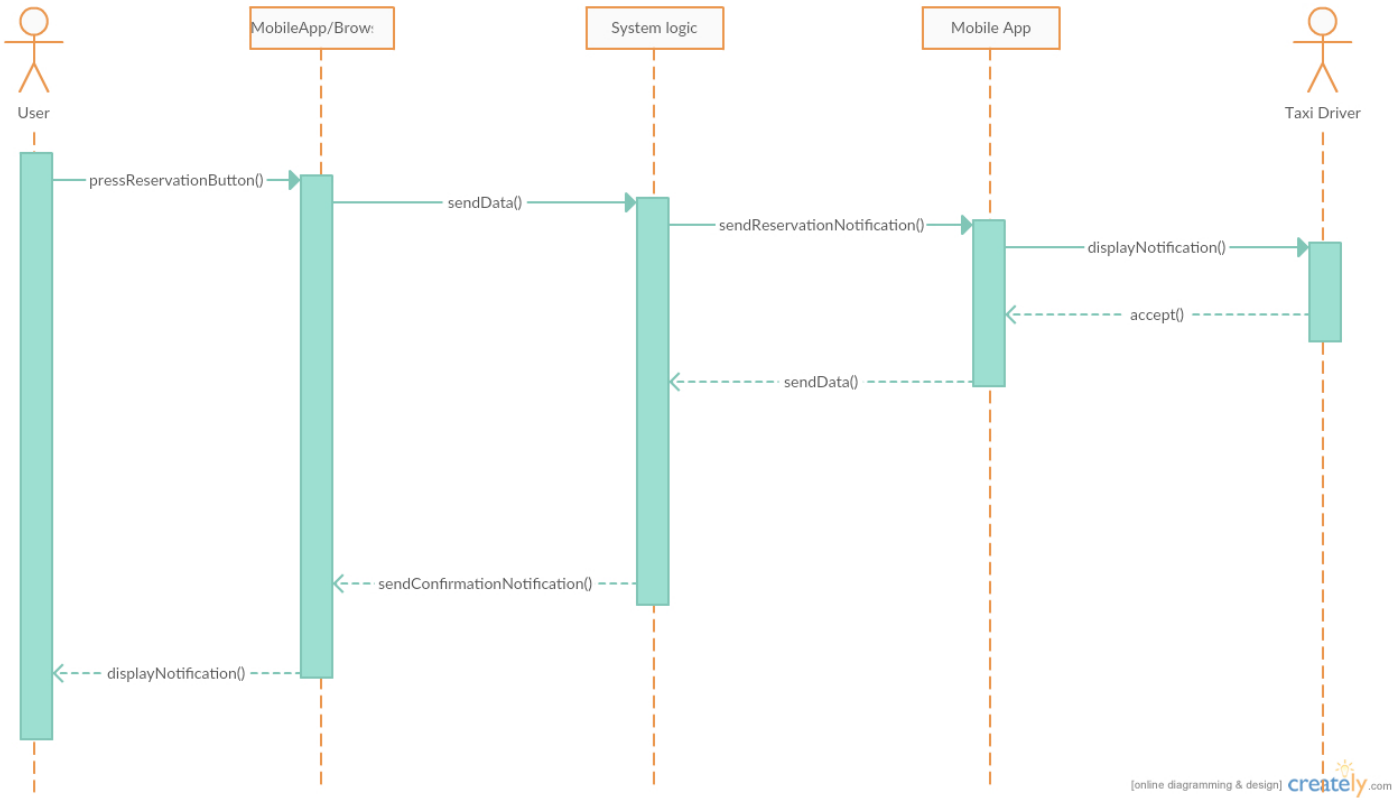

Figure 17: Sequence diagram for reservation operation

Queue operations

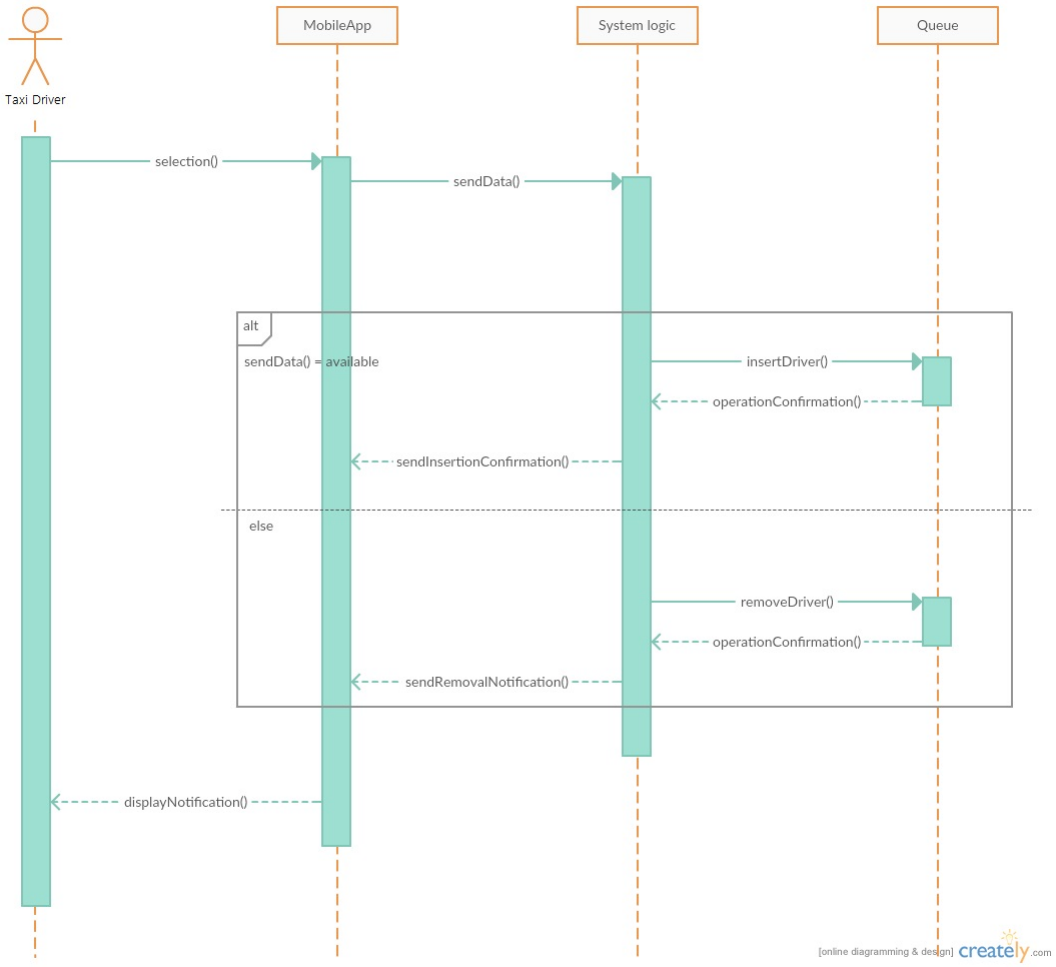

Figure 18: Sequence diagram for queue management functions

2.7 Component interfaces

Below are displayed the interfaces between the main components of the system (see fig. 6).

- **Interface name:** System interface
- **Provided by (component):** Application
- **Connected Components:** Web App, Mobile App
- **Offered methods:**
  - callRequest(): starts the process of a Call. (see fig 16).
  - reservationRequest(): starts the process of a Reservation. (see fig 17).
  - login(): allows the Login. (see fig 15).
  - registration(): allows the Reservation. (see fig 14).
  - availability(): manages the queue in respect of the driver's "available" setting. (see fig 18).
  - answerCall(): forwards the data of the Call to the designed Driver, starts a 1 minute timer and waits for an answer.
  - answerReservation(): forwards the data of the Reservation to the designed Driver and waits for an answer.

- **Interface name:** Data Access
- **Provided by (component):** Data
- **Connected components:** Application
- **Methods Offered:**
  - insertData(Object): performs a dataCheck(Object) and if it returns false, inserts the Object into the database, then returns true. Otherwise returns false.
  - removeData(Object): performs a dataCheck(Object) and if it returns true, removes the Object from the databse, then returns true. Otherwise returns false.
  - dataCheck(Object): if the Object is already present in the databse returns true. Otherwise returns false.

- **Interface name:** Browser UI
- **Provided by (component):** Web App
- **Connected components:** User
- **Methods Offered:**
  - pressRegistrationButton(): returns true when the Registration Button is pressed. False otherwise. Sends also the data inserted in the form to the Application.
  - pressLoginButton(): returns true when the Login Button is pressed. False otherwise. Sends also the data inserted in the form to the Application.
  - pressCallButton(): returns true when the Call Button is pressed. False otherwise.Opens also the Call form.
  - pressReservationButton(): returns true when the Reservation Button is pressed. False otherwise. Opens Also the Reservation form.
  - confirmCallButton(): returns true when the Call Button under the form is pressed. False otherwise. Sends also the data inserted in the form to the Application.
  - confirmReservationButton(): returns true when the Reservation Button under the form is pressed. False otherwise. Sends also the data inserted in the form to the Application.

- **Interface name:** Mobile GUI
- **Provided by (component):** Mobile App
- **Connected components:** User, Driver
- **Methods Offered:**
  - pressRegistrationButton(): returns true when the Registration Button is pressed. False otherwise. Sends also the data inserted in the form to the Application.
  - pressLoginButton(): returns true when the Login Button is pressed. False otherwise. Sends also the data inserted in the form to the Application.
  - pressCallButton(): returns true when the Call Button is pressed. False otherwise.Opens also the Call form.
  - pressReservationButton(): returns true when the Reservation Button is pressed. False otherwise. Opens Also the Reservation form.
  - confirmCallButton(): returns true when the Call Button under the form is pressed. False otherwise. Sends also the data inserted in the form to the Application.
  - confirmReservationButton(): returns true when the Reservation Button under the form is pressed. False otherwise. Sends also the data inserted in the form to the Application.
  - availability(): returns true when the I'm Available Button inside the Driver' s personal page is pressed.False otherwise.
  - answerCall(): returns true when the Accept Call Button inside the Driver' s personal page is pressed. False otherwise.
  - answerReservation(): returns true when the Accept Reservation Button inside the Driver' s personal page is pressed. False otherwise.

2.8 Other design decisions

One of the most important property of our application is the availability. The service that we want to provide must be available almost always to provide its functions to clients and taxi drivers. There are two identical copies of application server and the database. This is why we decided to have a copy for the application server and the database in order to have one copy of the logic part and one copy of the database for the data part. In fact, if the main server has a failure, the other server substitues the main one in order to maintain the system up. Regarding the database issue, while the first database in being used all the data registered are copied onto the second one (the right column in fig 19). As soon as a failure of the first database occurs the system switches machine and uses the secondary DB. When the main DB is fixed the first machine is updated with all the data registered by the second one during the down time. It is now

ready to work as soon as a failure of the right one happens. As stated in section 2.3 keep a system up 24 hours at 24 is not possible because it can happen, for example, a total failure of the system that disables both business servers or both databases for a while, but however we still consider a downtime of 1.6 hours per year negligible.

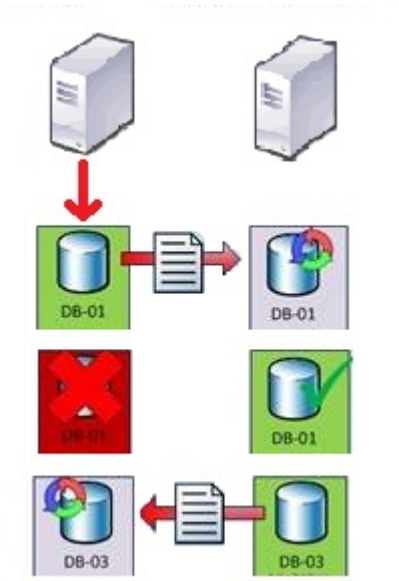

Figure 19: Solution of the availability issue regarding the database and the business logic.

3 Algorithm design

As already mentioned in section 1.5 "Document structure", this part is not present in the common design document but we will show an high level algorithm just to anticipate a basic function that in the implementation step will be described in a detailed way: the search of an element in a queue (flow chart in figure 20). This algorithm is an idea to solve the problem to find a specific taxi driver in the related queue in which it is located at a given time. In the future the developer can exploit this proposed algorithm or can change it, so he is left free to decide whether to use it or specify a more efficient one thought from him. The search is used because, if a taxi driver is working and wants to stop working for whatever reason, he communicates his will to the system via the botton "unavailable", see the section 3.1.1 "User interfaces" in the RASD document, that allows him to remove himself from the queue. To do so the system needs an algorithm that search the identifier of the taxi driver that want to stop working in the queue correspondent of the GPS position of the taxi, in

order to, once the position of the identifier has been found, thanks to the search algorithm, proceed with the elimination of it and the fix of the positions of the following elements in the queue in order to have a consistent queue (no hole in the middle).

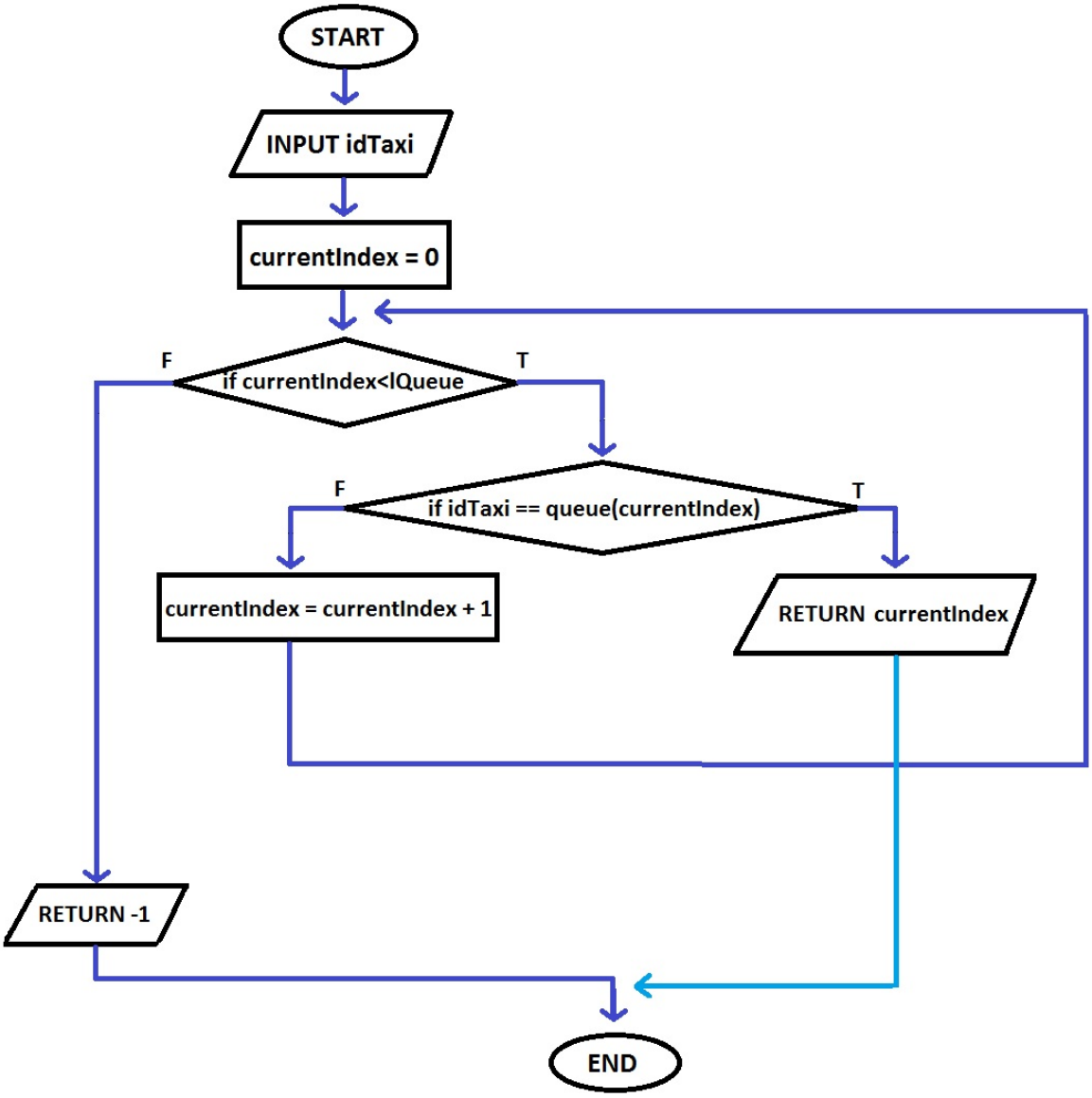

Figure 20: The search in a queue. The variables: 1) idTaxi represent the identifier of the taxi that I want to search. 2) currentIndex represent the current index that we are considering in queue. 3) Iqueue is the total number of elements in the queue.

4 User interface design

Refer to section 3.1.1 of the RASD document for the user interface design. There are presented mockups and details for both clients' and driver' s interfaces.

5 Requirements traceability

In the table below we show the mapping of requirements on the components of the system.

Requirement	Component	Comments
Registration into the system	• Data • System logic • Web app • Mobile app	The System logic takes the users' input inserted through the Web or Mobile app and gives them to the Data component to register them inside the database.
Login into the system	• Data • System logic • Web app • Mobile app	The System logic takes the users' input inserted through the Web or Mobile app and checks through the Data component if the data consitutes a valid login.
Call a taxi	• Application • Web app • Mobile app	The request is made through the Mobile or Web app, then the Application component manages the call forwarding, the queue, the answers from the drivers and the notifications.
Reserve a taxi	• Application • Web app • Mobile app	The request is made through the Mobile or Web app, then the Application component manages the answers from the drivers and the notifications.
Become available	• Application • Mobile app	Through the Mobile app a driver can choose to become available then the Application component manages the drivers' insertion in the queue.
Become unavailable	• Application • Mobile app	Through the Mobile app a driver can choose to become available then the Application component manages the removal of the driver from the queue and the positioning of the others drivers.
Answer an incoming call	• Application • Mobile app	Through the Mobile app a driver can answer to a call then the Application component manages his removal from the queue and the notifications.

6 Supporting informations

6.1 Glossary

- **Design space**: the space of possible designs that could be achieved by choosing different sets of alternatives.
- **Design pattern**: design solution to a recurring problem.
- **Thin client**: Is a computer or a computer program that depends heavily on another computer (its server) to fulfill its computational roles.
- **Model-View-Controller**: is an architectural pattern that separate the logic presentation of the data from business logic (the application logic that makes operational the application).

6.2 References

- **"Template for the design document"**: template for the redaction of the design document.
- **LaTeX documentation**: to support the redaction of this document.
- **LyX documentation**: to redact this document.
- Slides "**Design I -II**" by Raffaela Mirandola: to study about design and styles in general.
- "**Software Architectures and Styles**" by Damian A. Tamburri: to study about design and styles in general.
- Slides "**DesignPatterns**" by Carlo Ghezzi: to study about design patterns.
- **UML documentation**: to support the creation of diagrams.

6.3 Tools used

- **LyX**: used to write this document.
- **DropBox**: used to share materials between members of the group.
- **GitHub**: used to share materials between memebers of the group and to deliver this document.
- **Paint**: used to build the pictures.
- **Creatively.com**: used to create UML diagrams.

6.4 Workload

The time spent to redact this document, (included the time to think at a solution) is approx:

Lorenzo Federico Porro: 30 hours
Annalisa Rotondaro: 30 hours