SmartCityAdvisor

Design Document

Mirko Basilico, Federico Dazzan

May 29 2016

Table of Contents

1. INTRODUCTION

1.1 Purpose

This document describes the system architecture and design of the SmartCityAdvisor project. The architecture represents a high level view of the components of the system, while the design describes interactions between the different modules and how they will be mapped into programming elements in the future development process. This document will explain the architectural and design decisions and tradeoffs chosen in the design process and their justifications.

This document is intended to be used by all team members during the system development. It will also be used by the project supervisor to get insight into the project work. This is a technical document and it is not initially intended for the customers. However, technically educated customers may also get a general overview of the project using this document.

1.2 Scope

The system aims to offer a variety of services to the citizens of Milan, through a dedicated mobile application.

This document will describe all the components and all the interfaces connecting them, as well as the interfaces to external components or databases or users. The architectural descriptions will concern: component view, deployment view, runtime view, component interfaces, selected styles and patterns. These descriptions will consider all the main functionalities already explained in the RASD.

1.3 Definitions, acronyms, abbreviations

- **Tier:** It is a hardware level in a generic architecture
- **Layer:** It is a software level in a generic software system
- **GUI:** Graphical User Interface. It is the set of graphical elements such text, buttons and images which the user can interact with, usually abbreviated to **UI**

- **Thin Client:** It is a design style for the client in which the application running on the client contains the least business logic possible.
- **RDBMS and DB:** Relational Data Base Management System and Data Base Layer
- **Application Server:** the component which provides the application logic and interacts with the DB and with the front-ends
- **Back-end:** other term to identify the Application server
- **Front-end:** the components which use the application server services (e.g., the web front-end and the mobile applications)
- **Web Server:** the component that implements the web-based front-end. It interacts with the application server and with the users' browsers
- **API:** Application Programming Interface
- **Java EE:** Java Enterprise Edition
- **JDBC:** Java Data Base Connectivity
- **JSON:** JavaScript Object Notation, it is a lightweight data-interchange format (attribute–value pairs) that usually replaces XML for asynchronous browser/server communication
- **JSF-JSP:** Java Server Faces – Java Server Pages

1.4 Reference documents

Specififcation document: New Project April 2016-2.pdf

RASD document:rasd.lyx

IEEE Std 1016-2009, IEEE Standard For Information Technology - System De- sign - Software Design Descriptions.

1.5 Document structure

This document is structured in four main parts:

- **Architectural Design:** this section shows the main components of the systems with their sub-components and their relationships, along with their static and dynamic design. This section will also focus on design choices, styles, patterns and paradigms

- **Algorithm Design:** this section presents and discusses in detail the algorithms designed for the system functionalities
- **User Interface Design:** this section provides an overview on how the UI(s) of the system will look like
- **Requirements Traceability:** this section shows how the requirements defined in the reference RASD map into the design elements defined in this document

2. ARCHITECTURAL DESIGN

2.1 Overview

To develop this project we decided to use the Java Enterprise Edition architecture, that is four-tiered, with GlassFish as Application Server and MySQL as RDBMS. Moreover, we decided to decouple the business logic from the view, using a RESTful architecture, since we have completely different interfaces to deal with (mobile application and web application for users):

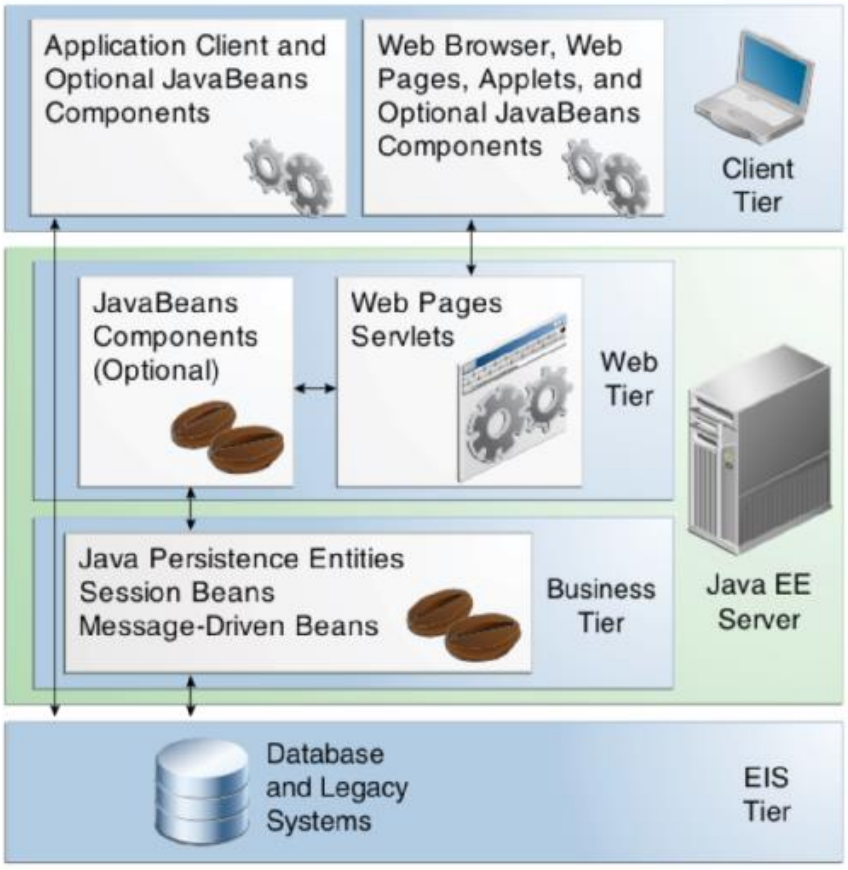
• **Client Tier:** it contains Application Clients and Web Clients and it is the layer that interacts directly with the actors. As far as the web application part is concerned, the client will use a web client to access pages, that consists of dynamic pages with markup language (like HTML and XML) and a web browser (Google Chrome, Mozilla Firefox) that renders the pages received from the web server. The mobile application for users, instead, will be developed using a native language such as Objective C or Swift for iOS and Java for Android or a framework like Apache Cordova (HTML/CSS/JavaScript) for a cross-platform single-page application. All of them will require resources, obtained by calling specific APIs

• **Web Tier:** it contains web components that can be Servlets or Dynamic Web pages created with JSF and/or JSP technology. It is possible to use some Java Beans components to send input data (in JSON format) made by users to other beans running in the business tier and forward the response to the client. Moreover, It is the layer that is exposed to the outside: client requests are processed by a dispatching component (WebController) which decides which controller to execute. Every controller uses the components it needs to perform an action

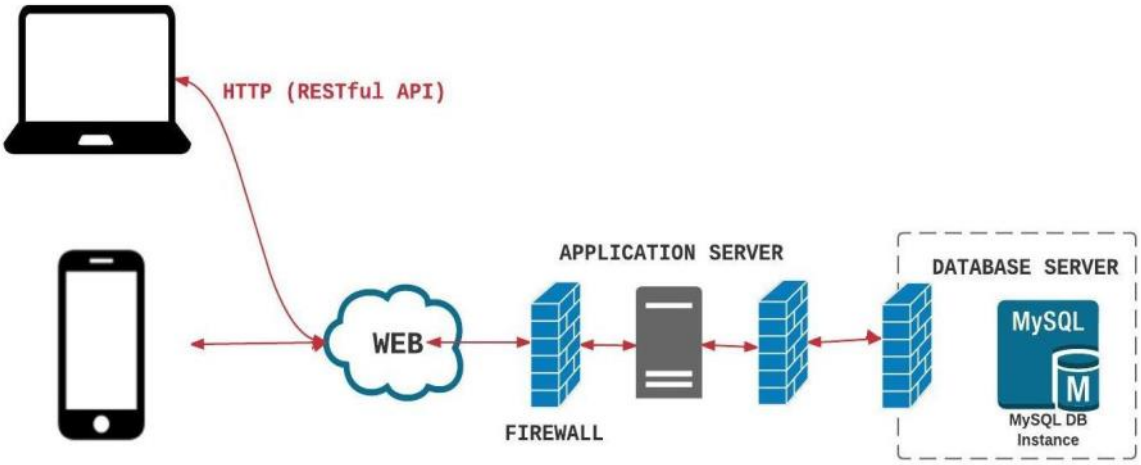
• **Business Tier:** essentially it is the logic part that solves the needs of a particular business domain. It consists of Enterprise Java Beans (EJB), that process data coming from the client and send it to the Enterprise Information System tier for storage or the other way around (using JPA), and Java Persistence Entities (Session Beans and Message Driven Beans). In our case, Stateless Session Beans

are used since they do not maintain a conversational state with the client. When a client invokes the methods of a stateless bean, the bean's instance variables may contain a state specific to that client but only for the duration of the invocation. When the method is finished, the client-specific state is not retained. Except during method invocation, all instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client. Because they can support multiple clients, stateless session beans can offer better scalability for message-driven applications that require large numbers of clients (like ours) than the stateful ones. Nevertheless, our architecture is RESTful, so it must be stateless by definition

• **Enterprise Information System (EIS) Tier:** it includes enterprise infrastructure systems with the DBs in which data concerning citizens, hospital,parking and transport station are stored and retrieved. Java EE application components need access to this tier for database connectivity (JDBC)



This design choice also makes it possible to deploy the application server and the web server on different tiers, improving scalability where many web servers talking to a single application server are desired. The interactions between the main architectural components are shown in the hardware representation, in which the web server is omitted for simplicity since it can be included in the application server (detailed deployment view in chapter 2.4). They are all synchronous and multi-threaded (the more the users of the system, the more the instances). Security measures are present to encrypt communications and protect the data from unauthorized access and malicious users.



HARDWARE REPRESENTATION

Our RESTful architecture permits to have a single back-end which can be used from all client applications, that are the mobile applications (regular users) and the web application for the government. Every client makes an API call to the web-server and handles the answer in different ways according to where the request came from. It is important to state that mobile applications and web application have a different view logic.

As far as security is concerned, every HTTP request must have an associated authentication-authorization token (created at login time to identify the client). In fact, the request is sent by the "Dispatcher" component to the controller if and only if the associated token is correctly validated by the "SecurityManager" component. The controller, then, will satisfy the desired function using other components. Authentication is a process in which the credentials provided are compared to those on the database. If the credentials match, the process is completed with the recognition of the client (with

their type: citizen and government), that is then granted authorization to access the functionalities of the system specifically designed for them. The produced response is a JSON string that all client applications can handle.

One of the advantages of Java EE is represented by the fact that many services have been produced by other developers and can be used in our application: for instance, JAX-RS (Java API for RESTful Web Services) allows to create APIs following the RESTful paradigm, in order to extend the functionalities provided by our system.

2.2 High level components and their interactions

Dispatcher

This component dispatches all the requests that come from a citizen. When a request is received the component checks if the request is valid and, after that, if it is, the request is forwarded to the correct component. Instead, if the request is not valid, the dispatcher returns an error message to the user. When it receives a response from a component, it sends it to the correct user.

SecurityManager

The unique function of this component is to control if a user's request is valid. After receiving a request, this component has the responsibility to check whether who made it is allowed to do that or not.

AccountManager

This component manages the requests of the user that deals with account management. When a user logs in this component creates a token and stores it in the database. Instead, when a user logs out this component removes the relative token from the database.

HospitalManager

This component is in charge of managing all the requests made by a user concerning the emergency rooms.

ParkingManager

This component is in charge of managing all the requests made by a user concerning the parking facilities.

PublicTrasnport

This component is in charge of managing all the requests made by a user concerning the public transport system.

TrafficManager

This component manages all the requests that could lead to a limitation of the city traffic.

These requests could come from:

- Automated CO2 Threshold signal sent by DCC
- Manual request by the City administration

DCCManager

This component manages all the request that have to sent to DCC, through the specific API.

TCCManager

This component manages all the request that have to sent to TCC, through the specific API.

DataManager

This component manages all the operations regarding the data base (Insert, Update, Delete).

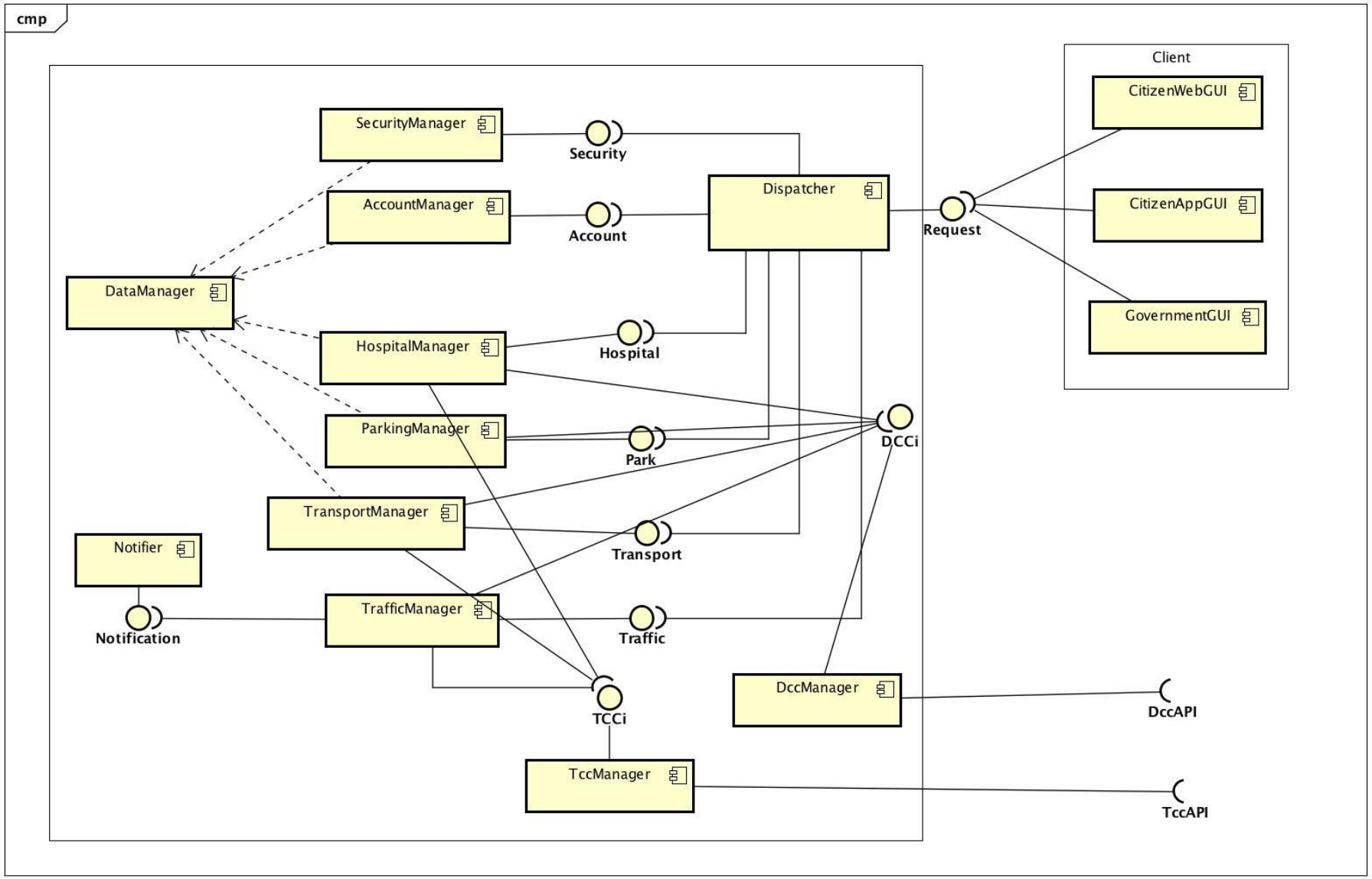
CitizenGUI / GovernmentGUI

These client-side components manage the visual representation of the user interface

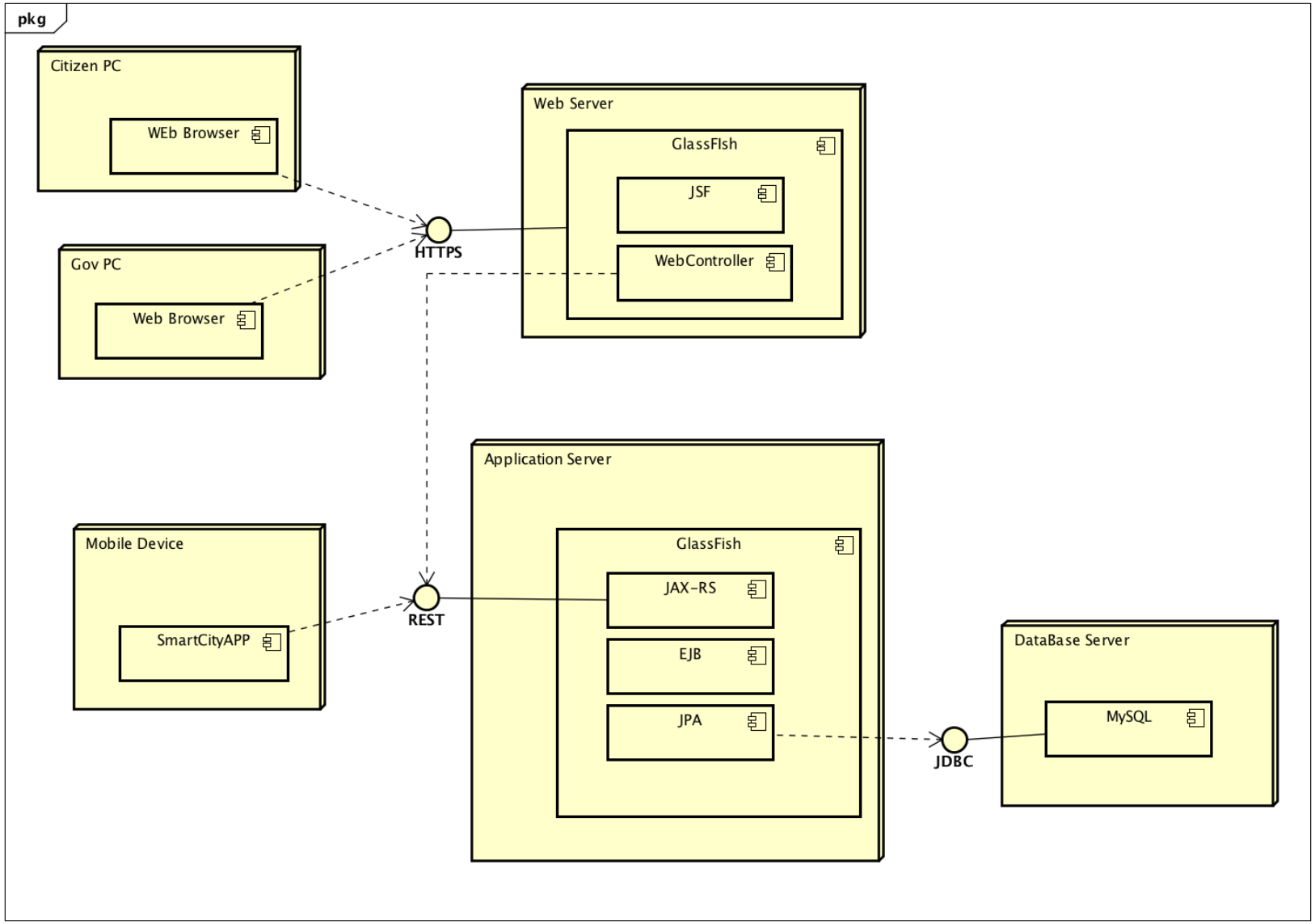
Notifier

This component manages all the notifications that will be sent to the mobile app.

2.3 Component view



2.4 Deployment view



2.5 Runtime view

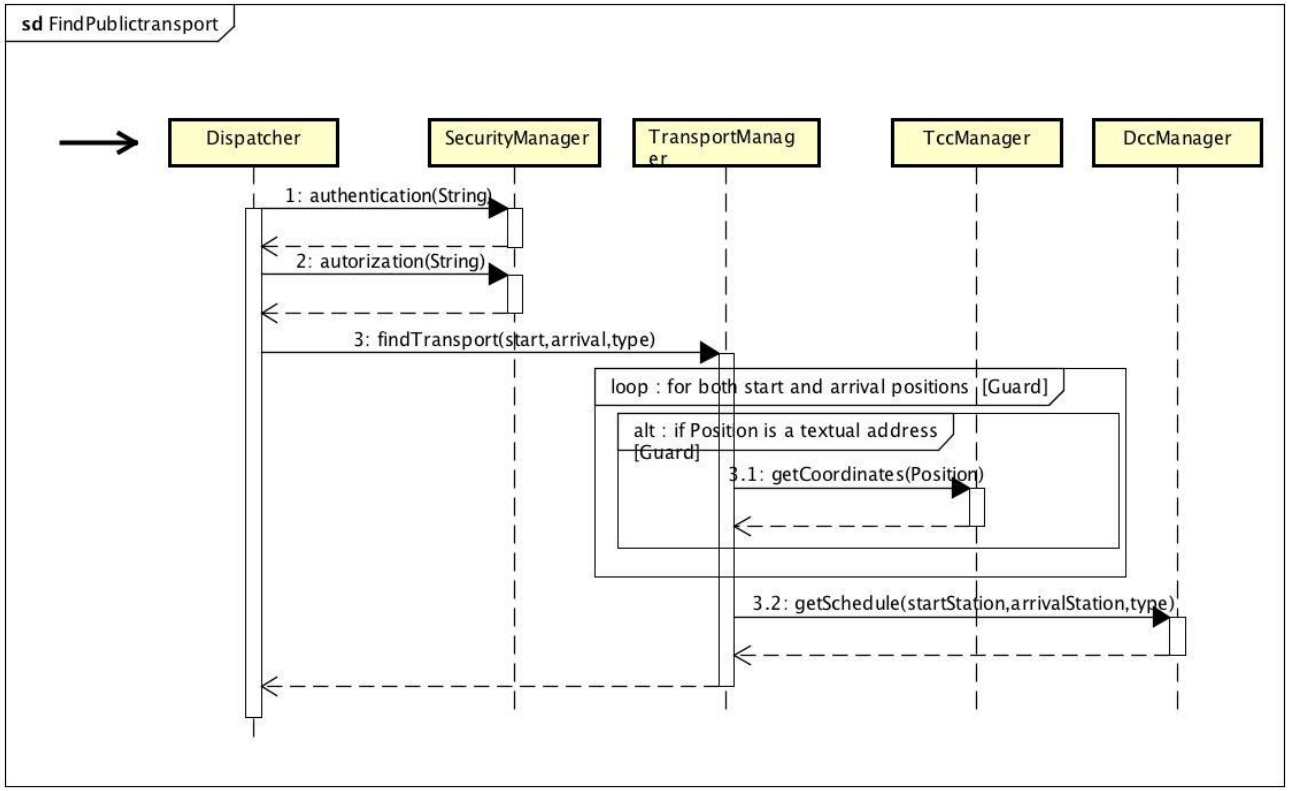


Figura 1: Find public transport sequence diagram

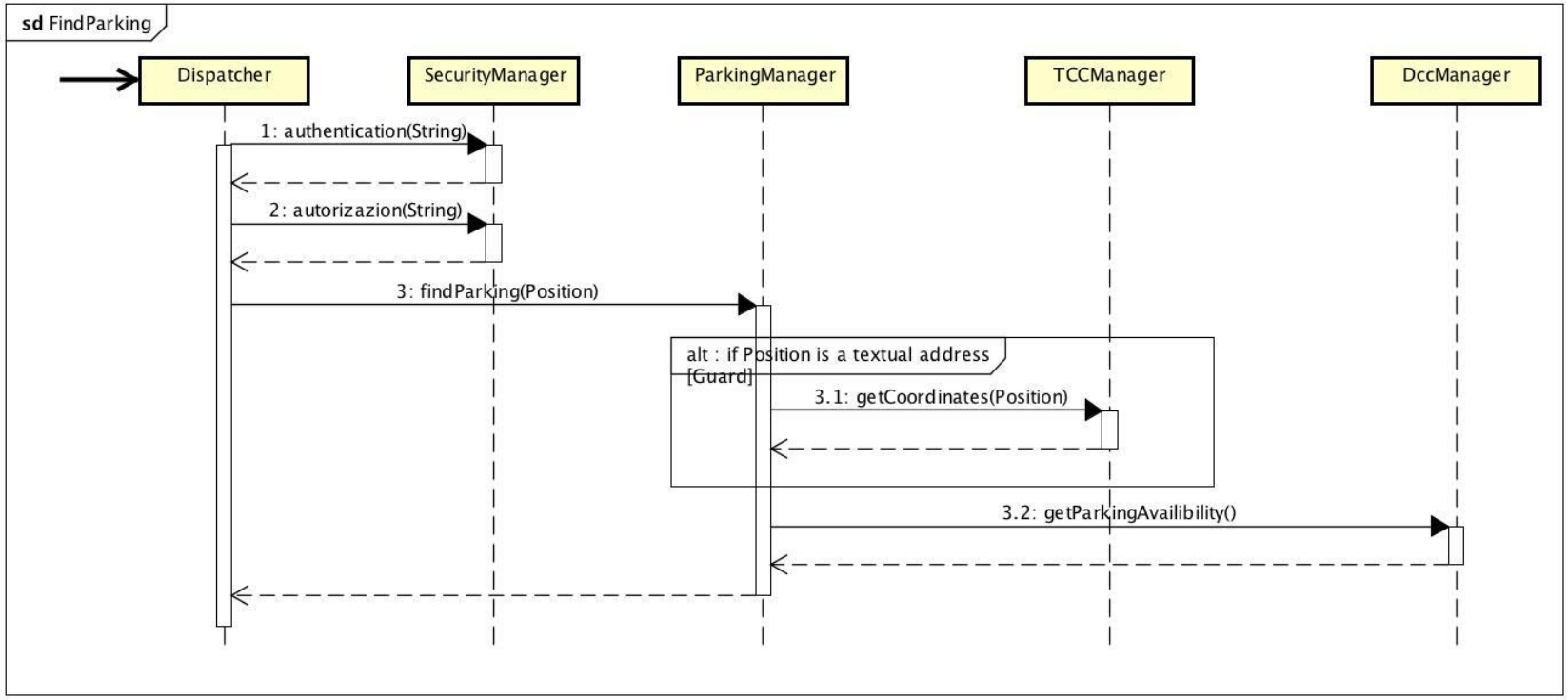


Figura 2: Find parking sequence diagram

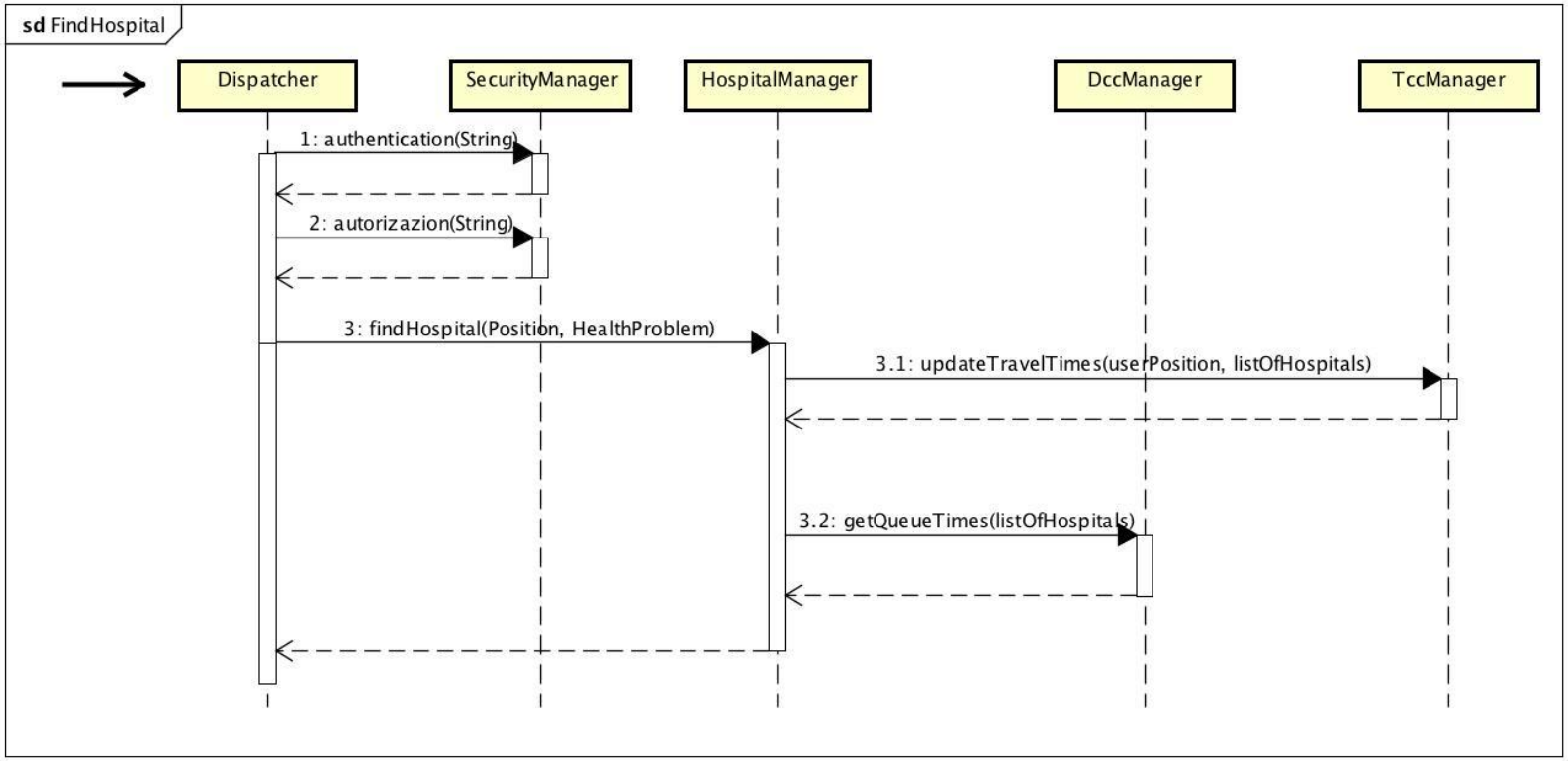


Figura 3: Find hospital sequence diagram

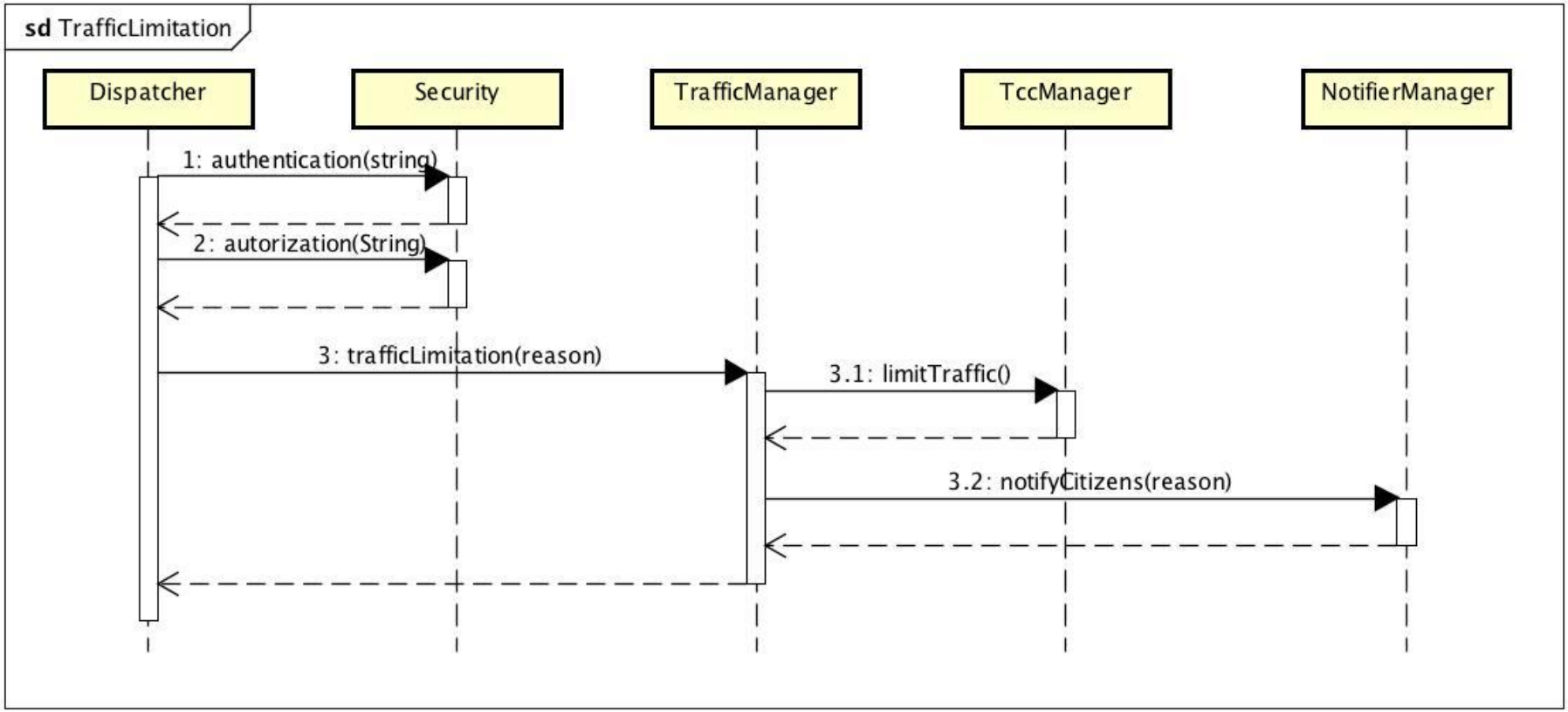


Figura 4: traffic limitation sequence diagram

2.6 Component interfaces

SecurityManager interface

```
Interface Security{
        public Boolean authentication(String token);
        public Boolean authorization(String token);
}
```

AccountManager interface

```
Interface Account{
        public Response signIn(String s);
        public Response signOut(String s)
}
```

TrafficManager interface

```
Interface Traffic{
        public void trafficLimitation(String reason)
}
```

HospitalManager interface

```
Interface HospitalI{
        public Hospital findHospital(Position userPosition, String healthProblem);
}
```

ParkingManager interface

```
Interface ParkingI{
        public Parking findParking(Position userPosition);
}
```

TransportManager interface

```
Interface TransportI{
public TransportInformation findTransport(Position start, Position arrival, String transportType);
}
```

DCCManager interface

```
Interface DCCi{
        public ArrayList<Parking>getParkingAvailability();
        public void updateQueueTimes (ArrayList<Hospital> compatibleHospitals);
        public Schedule getSchedule(Station startStation, Station arrivalStation, string type);
}
```

TCCManager interface

```
Interface TCCi{
        //add to position the coordinates associated the textual address
        public void getCoordinates(Position position);
        public void limitTraffic();
        public void updateTraveltimes(Position userPosition, ArrayList<Hospital> hospitals);
}
```

NotifierManager interface

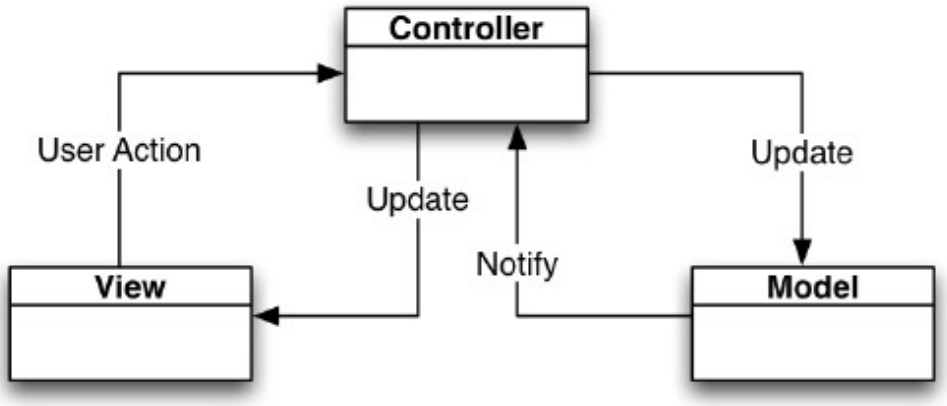
```
Interface NotifierI{
        public void  notifyCitizens(String reason);
}
```

2.7 Architectural styles and patterns

The following design patterns have driven the design process of this project:

• **MVC design pattern:** Model-View-Controller. This software architectural pattern divides the business data, the user interface (or the interface between systems) and the core modules that run the business logic, so as to separate internal representations of information from the way they are presented

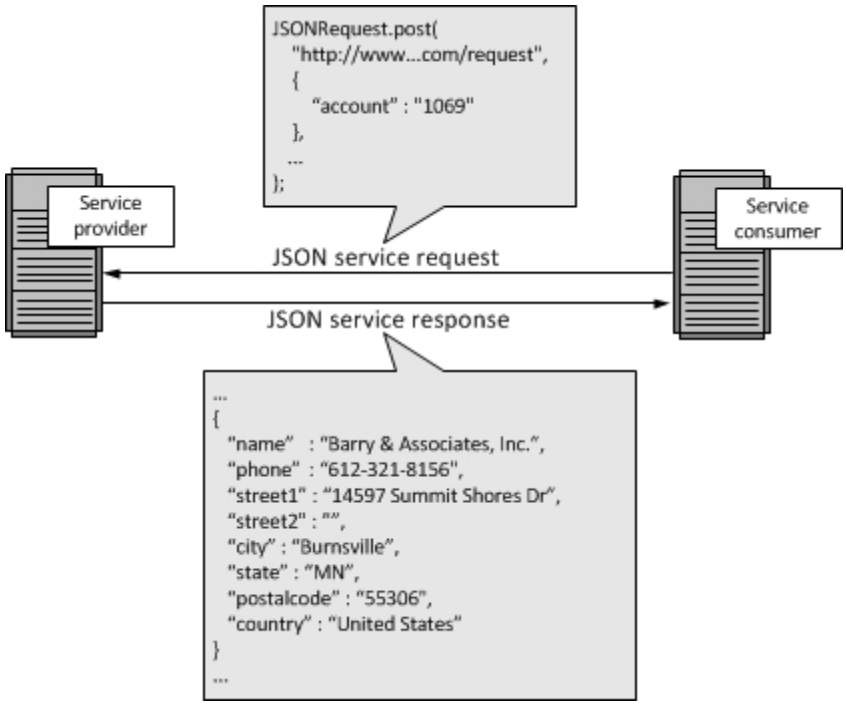
to or accepted from the user. In SmartCityAdvisor it will be applied to the client side (Mobile and Web applications).



• **Thin Client Server architecture:** The client–server model is a distributed application structure that divides tasks between the providers of a resource or service (servers) and service requesters (clients). The clients in our architecture are both thin and fat, because some contain only an interaction/visualization layer and some also contain the business logic of the application, even if most of that is concentrated in the application server

• **RESTful architecture:** Representational State Transfer (REST) is a style of architecture based on a set of principles that describe how networked resources are defined and addressed. An application or architecture considered RESTful is characterized by these properties: state and functionality are divided into distributed resources; every resource is uniquely addressable using a uniform and minimal set of commands (typically using HTTP commands of GET, POST, PUT, or DELETE over the Internet); the protocol is client/server, stateless, layered, and supports caching.

The following figure illustrates the usage of REST for Web Services with JSON as data-interchange format:



3.ALGORITHM DESIGN

Here below are reported some of the algorithms we developed for our system (not managing database errors).

TrafficManager algorithms

The following function of the TrafficManager is called by the Dispatcher when the DCC or the government of Milan ask our system to limit the traffic in the center.

```
public void trafficLimitation(String reason){
        //ask to DCC to manage the limitation of the traffic in the center
        dcc.limitTraffic();
        // ask to notifier manager to warn all the registered citizens that the traffic is closed
        notifier.notifyCitizens();
}
```

HospitalManager algorithms

The following function of the HospitalManager is called by the Dispatcher when a citizen use the mobile or the web application to use the "find emergency room" feature of our system.

```
public Hospital findHospital(Position userPosition, String healthProblem){
        ArrayList<Hospital> compatibleHospitals = findHospitalsGivenAnHealthProblem(healthProblem);
        //updateTravelTimes() updates the ETA values associated to the hospitals in the list that is indicated
        //as parameter of the function
        tcc.updateTravelTimes(userPosition, compatibleHospital);
        //updateQueueTimes()  updates the queue time values associated to the hospitals in the list that is
        //indicated as parameter of the function
        dcc.updateQueueTimes(compatibleHospital);
        //calculateBestHospital() finds the best hospital among those in the list that is indicated as
        //parameter, considering ETA and queueTime values
        Hospital bestHospital = calculateBestHospital(compatibleHospital);
        return bestHospital;
}
```

ParkingManager algorithms

The following function of the ParkingManager is called by the Dispatcher when a citizen use the mobile or the web application to use the “find parking” feature of our system.

```
public Parking  findParking(Position userPosition){
        if (userPosition.getCoordinates == null)
                //get coordinates() adds to position the coordinates associated the textual address
                tcc.getCoordinates(Position);
        ArrayList<Parking> parking = getParkingAvailability();
                //findClosestParking return the parking closest to userPosition among those in the list that is
                //indicated as parameter of the function
        Parking bestParking = findClosestParking(userPosition, parking);
        return bestParking;
}
```

TransportManager algorithms

The following function of the TransportManager is called by the Dispatcher when a citizen use the mobile or the web application to use the “find public transport” feature of our system.

```
public TransportInformation findTransport(Position start, Position arrival, String transportType){
        if (start.getCoordinates == null)
                //get coordinates() adds to position the coordinates associated the textual address
                tcc.getCoordinates(start);
        if (arrival.getCoordinates == null)
                tcc.getCoordinates(arrival);
        //findClosestStation() returns the station of the indicated transport type closest to the indicated
        //position
        Station startStation = findClosestStation(start, transportType);
        Station arrivalStation = findClosestStation(arrival, transportType);
        //getSchedule requires to DCC the schedule of the first train/bus/tram(accordin with the
```

```
        //indicated transport type) that will arrive at the indicated start station
        Schedule schedule = dcc.getSchedule(startStation, arrivalStation, transportType);
        //calculateTransportInfo() returns a TransportInformation object, that containsall the information
        //that our system has to show to the citizen
        TransportInformation info = calculateTransportInfo(startStation, arrivalStation,schedule);
        return info;
}
```

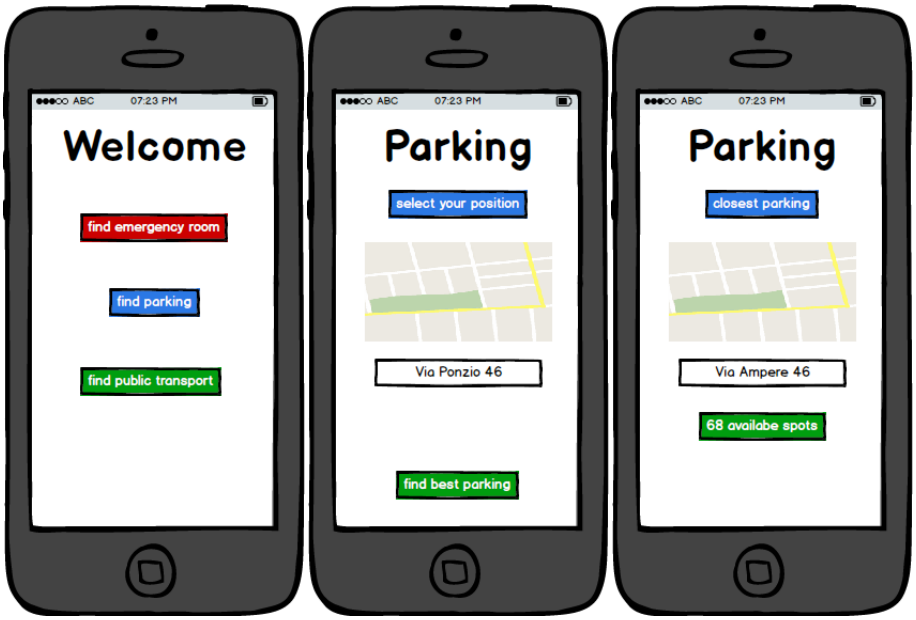
4. USER INTERFACE DESIGN

In this section we will show the UI/UX that our system will provide to the users, for both the mobile application and the web application.

We will do this through self-explanatory mockups

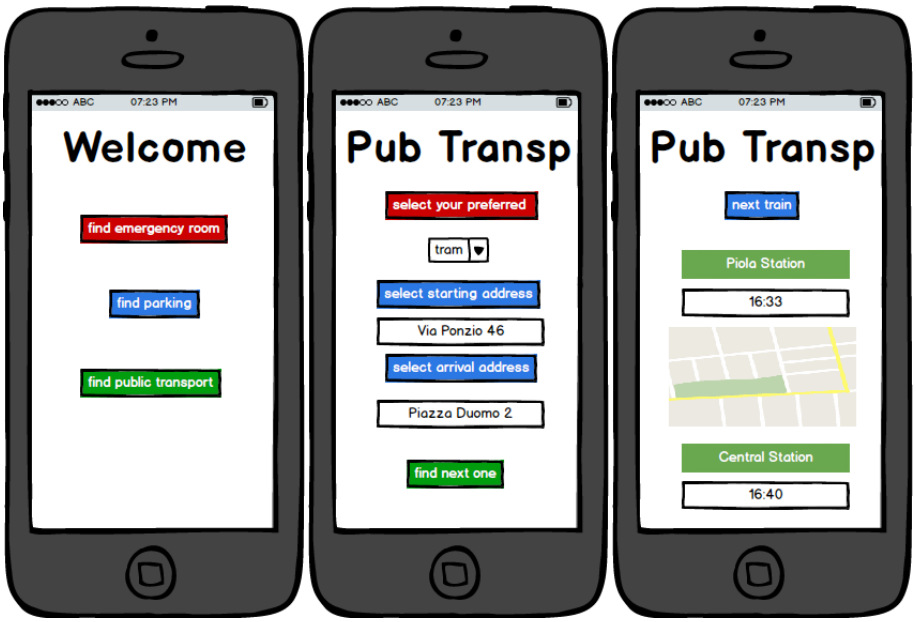
Login->Registration->Homescreen

Homescreen->Find Parking->Confirmation

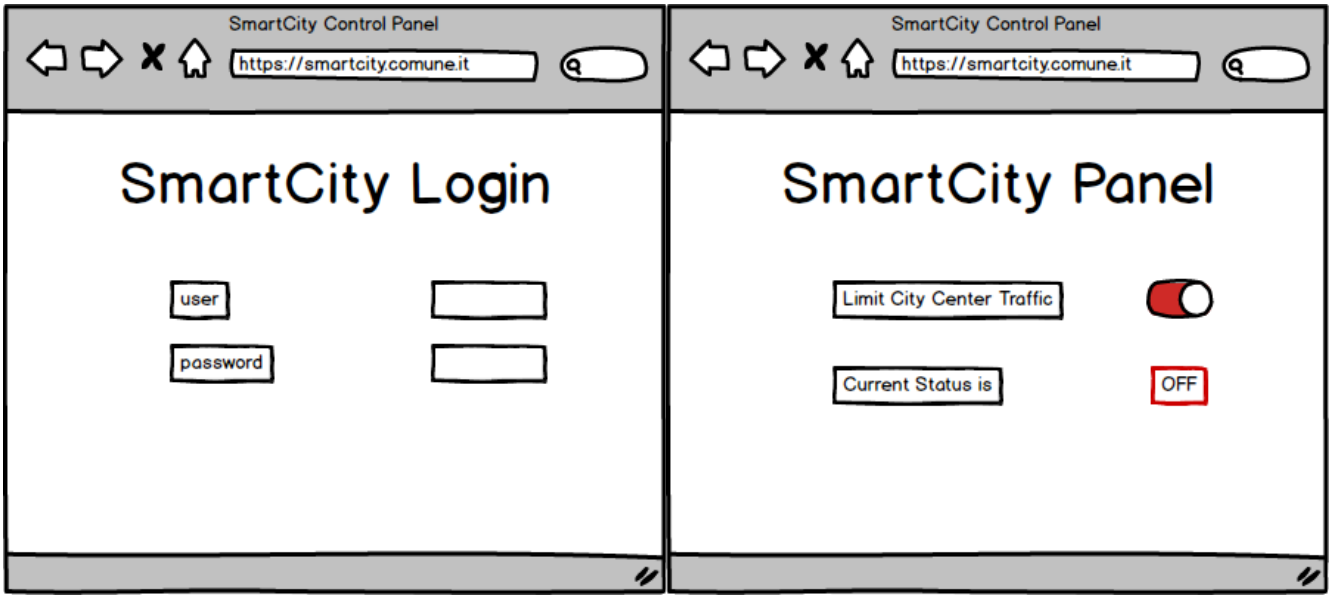


Homescreen->Find Emergency Room->Confirmation

Homescreen->Find Public Transport -> Confirmation



Government Web Application



5. REQUIREMENTS TRACEABILITY

ACTOR:

REQUIREMENT → COMPONENT

GUEST:

Register to the system → SecurityManager

USER:

Log in into the system → SecurityManager

Find best hospital → HospitalManager

Find Parking → ParkingManager

Find Public Transport → PublicTransport

GOVERNMENT:

Limit Traffic manually in the City centre→ TrafficManager

Limit Traffic automatically in the City centre → TrafficManager

6. USED TOOLS

- **Microsoft Word:** to redact and format the document
- **Balsamiq Mockups**: to create the user interface mockups
- **Astah Professional:** to create component, deployment and sequence diagrams
- **Lucid Chart:** to create the architectural model

7. HOURS OF WORK

We mostly worked remotely through voice and text chat, sometimes with screen sharing tools. We equally divided the workload for a total of 60h.

8. REFERENCES

- Slides of the Software Engineering 2 course
- Design Document Template
- http://www.service-architecture.com
- https://docs.oracle.com/javaee/