POLITECNICO
MILANO 1863

Software Engineering 2

Design Document

SAFESTREETS

Amorosini Francesco
Casali Alice
Fioravanti Tommaso

8 December 2019

Introduction

1.1 Purpose

While the RASD presented a general view of the SafeStreets appliction and its features, this document aims to further analyze the system's design and architecture, as well as describing for each of its components their runtime behaviour, integration, interfaces, implementation and testing plans. This document is mainly intended to be used by the test and development teams as a guidance in the development process, but also to prevent structural degradation during maintainance and extension phases. Nonetheless, the document is also addressed to all the stakeholders who are interested in supervising the development process.

1.2 Scope

SafeStreets: an application that aims to improve the safety of urban areas by giving its users the possibility to report traffic violations to authorities. Users are logged in either as citizen, those who report the violations, and authorities, those who are notified about newly reported violations and are supposed to take action on them.
The system is in charge of collecting all the Reports, storing them, and notifying the authorities about them. The stored Reports are then used to build statistics, find unsafe areas, and compute suggestions on how to improve the safety of such areas. The system may also communicate with local Municipalities' Systems in order to retrieve information about accidents and iussued traffic ticket: in this case some of the above functions are enhanced, and some new functions are enabled (g.e computing statistics on traffic ticked).

Further information about the scope of the application can be found in the Chapter 1 of the RASD.

1.3 Definitions, Acronyms and Abbreviations

1.3.1 Definitions

- *Client*: a piece of computer hardware or software that accesses a resource or a service made available by a Server.
- *Server*: a device or a computer program that provides resources or functionalities to other programs or devices.
- *Reverse Proxy*: particular proxys that are responsible of forwarding requests to one or more Servers which will handle it.
- *Firewall*: a network security systems that monitors incoming and outcoming network traffic, applying predefined predefined security rules.

1.3.2 Acronyms

- **RASD**: *Requirement Analysis and Specification Document*, the document in which all the requirements and goals of the application are throughly described.
- **API**: *Application Programming Interface*, interface, or communication protocol between Client and Server intendend to simplify the building of the Client-side software.
- **GPS**: *Global Positioning system*, technology widely used to get the user's position.
- **DBMS**: *Data Base Management system*, software that provides organized space memory to store information.
- **OCR**: *Optical Character Recognition*, software dedicated to the detection of characters contained in a document and to their transfer to digital text that can be read by a machine. In this context, OCR will be used to read license plates.
- **UML**: *Unified Modeling Language*, a standard visual modeling language intended to be used for analysis, design, and implementation of software-based systems.
- **MVC**: *Model View Controlles*, a software design pattern commonly used to provide user interfaces.

1.3.3 Abbreviations

- [R$_i$]: i-th requirement.

1.4 Reference Documents

- Specification document: *SafeStreets Mandatory Project Assignment.pdf.*
- Requirement Analysis and Specification Document: *RASD.pdf.*

1.5 Document Structure

This document is presented as it follows:

1. **Introduction** presents a general overview, the scope and the purpose of the document.
2. **Architectural Design** shows the main components of the system and their relationships. This section will also discuss the architectural choices of the design process.
3. **Algorithm Design** presents and discusses the algorithms that will enable the system's functionalities.
4. **user Interface Design** provides some further details on the user interface defined in the RASD.
5. **Requirement Traceability** maps all the functional requirements defined in the RASD over the components that will accomplish them.
6. **Implementation, Integration and Testing Plans** shows the order in which the implementation and the integration of the components will occur, and how the testing phase will be carryed out.
7. **Effort spent** displays the time spent writing this document by each member of the team.

Architectural Design

2.1 Overview

In this chapter the architectural structure of the system will be discussed at multiple levels of abstraction. A high-level view of the components and their interactions is represented in Figure **??**. The details will be explained in the next sections.

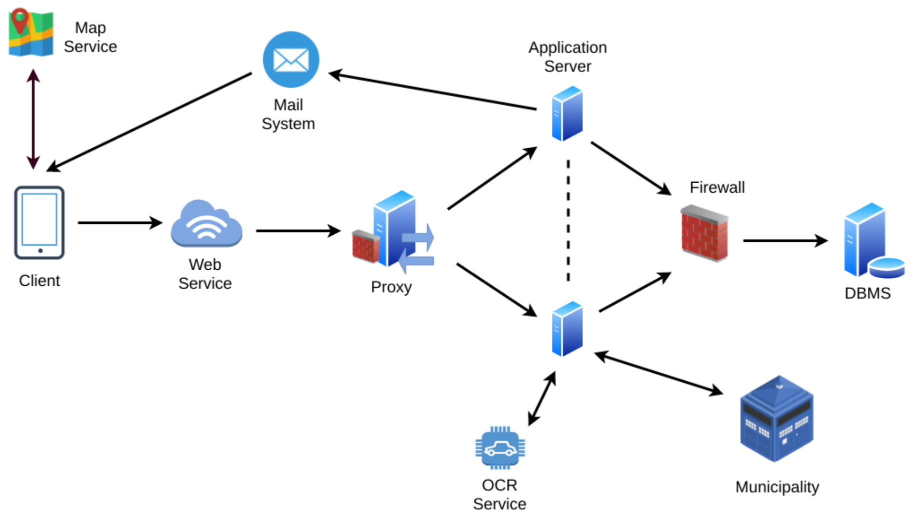

Figure 2.1: High-level overview of the system.

2.2 Component view

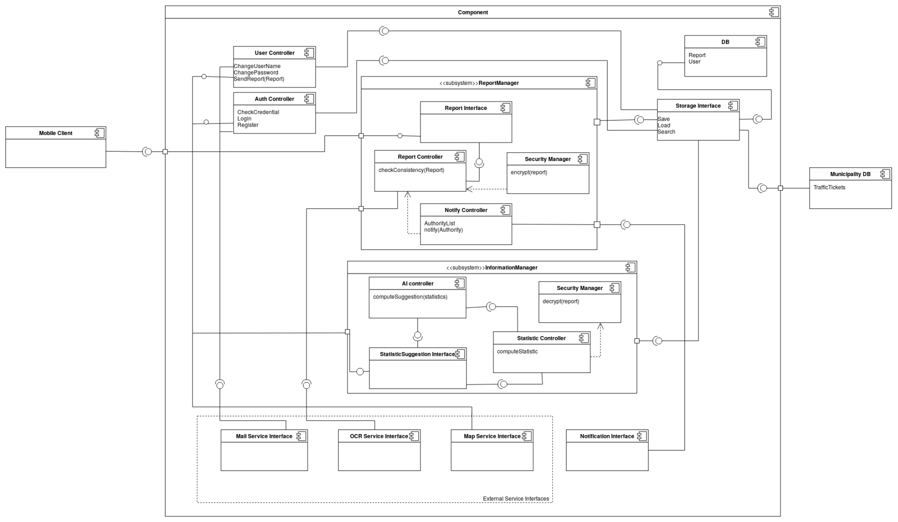

Figure 2.2: Component diagram of the system.

The UML component diagram aims at capturing the internal modular structure of the components, showing how they are connected together in order to form larger components. Components are wired together by using an assem- bly connector to connect the required interface of one component with the provided interface of another component. Below is a description of each component:

- **Mobile Client**: This component represent the client machine that access to the functionalities of the entire system. It is implemented as thin client as explained below.
- **User Controller**: This component takes care of all the operation related to the user data. It exposed methods to change account credentials and to send the new report of the user. Furthermore, it manages the data stored in the DB using the Storage Interface.
- **Auth Controller**: This component takes care of all the authentication-related operation: it is responsible both for the log in and the registration of the User and uses the *Storage Interface* to retrieve and store data in the DB. It also uses the *Mail Service Interface* in order to communicate with the Mailing Service to send registration emails.
- **Report Manager**:This component is a *subsystem* component, indeed it is composed of several subcomponents:

- **Report Interface**: this component exposes to the User the interface of the report's form to be compiled.
- **Report Controller**: this component aims to check the consistency of the report sent. To do this it uses the Report Interface to know how the report form is and how the report sent has been compiled.
- **Notify Controller**: the role of this component is to notify the correct authorities based on their municipality, using his *AuthorityList*. It depends on the Report Controller, in the sense that it will notify the authorities only if the check of the report is successful and the type of the report is *Traffic Violation*.
- **Security Manager**: this component takes care of the security side: as specified in the Security section of the RASD, before being stored, the report is encrypted and then stored in the DB. This task is accomplished by this specific component, that also depends on the Report Controller due to the fact that if the report doesn't pass the check verification, it will never be stored and therefore never encrypted.

2.3 Deployment view

In this section it is described the deployment view of the components inside the system. The deployment diagram describes the distribution of components of the hardware.

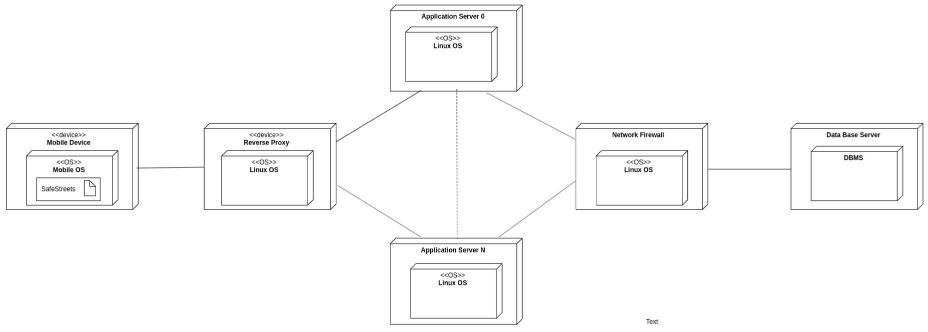

Figure 2.3: Deployment diagram of the system.

The system is composed of a multitier architecture, each specific role is clarified below.

- **Client**
  The first tier contains the client mobile machines which have installed SafeStreets application to access all its functionalities.

- **Reverse Proxy**
  The second tier contains a reverse proxy to implement load balancing on the several requests to access the application servers. Further, it is a cacheble component who can speed-up the most frequent requests. We decide to use a Linux machine for safety and semplicity reasons.

- **Application servers**
  This is the middleware level of the application where all the computations happen. The servers are distributed to increase the scalability of the network, they are also part of the second tier.

- **Firewall**
  The access to the Database is protected from a firewall to avoid unauthorized accesses to sensible data.

- **Database Server**
  This is the last layer of the architecture. All the data are stored here structured in a relational DBMS.

2.4 Runtime view

2.5 Component interfaces

2.6 Selected architectural styles and patterns

The following architectural patterns are used to build the structure of the system in order to provide all the services of SafeStreets application.

Client-Server Architecture

Client-Server architecture is a computing model that features two roles: a Server that hosts, delivers and manages most of the resources and services, and a Client which exploits them.

Motivations

This structure provides several advantages:

- Scalability and Mantainability: it is possible to repair or add more resources to the architecture without significative service interruptions.
- Security: the server is able to manage what levels of access each user can have with respect to specific resources.

Three-tiered Architecture

This type of architecture is a kind of Client-Server paradigm where three tiers are phisically separated:

- The *presentation tier* is the top-most level of the architecture, which provides an interface that users can use to directly access the application. It is the top-most tier and the only one accessible from the Client.
- The *application tier* runs the business logic of the application and executes functions that elaborate data. A Reverse Proxy is needed to handle the Client requests and to balance the workload, the requests are forwarded to the application serves in order to provide the right data.
- The *database tier* includes the data persistence mechanisms and the data access layer that encapsulates the persistence mechanisms and exposes the data.

Motivations

A multi-tier application architecture provides a model with several advantages: in this way developers can create flexible and reusable applications that can be modified, enhanced or maintained just by operating on a specific layer, instead of reworking the entire application. Furthermore, a general multitier architecture can also help improve the development efficiency by allowing teams to focus on their core competencies.

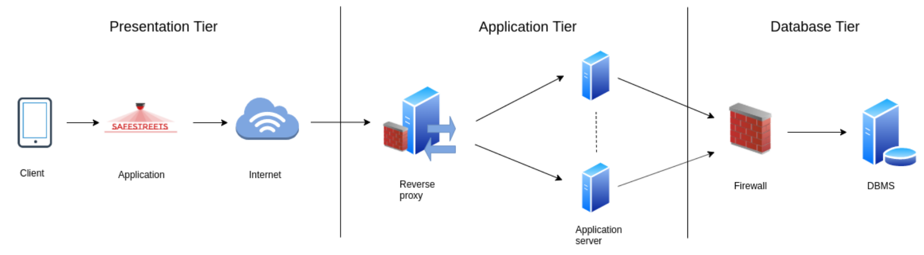

Figure 2.4: Three-tier architecture schema.

Thin Client

A thin Client is a lightweight computer which does most of the computation in a remote Server. In this paradigm in fact, since the Servers take care of several duties such as storage of data and performing calculations, the Client does not need to have a large memory or powerful computing capabilities to run the application.

Motivations

The application is thought for mobile phones which do not have great computing power or very large memories. Thanks to a thin client a internet connection is virtually the only requirements to use the application. Furthermore, this architecture simplify the front-end implementation by shifting most of the execution to the servers.

MVC Design Pattern

Model-View-Controller is a software design pattern commonly used to develop user interfaces. Indeed, this pattern is used for the front-end implementation of the application. It divides the program logic in three interconnected elements: *model* that directly manages data and rules of the application, *view* which handles any representation of data, *controller* that accepts input and converts it to commands for the model or view. This mechanism is used to separate internal representation of information from the ways information is presented to and accepted from the user.

Motivations

MVC allows full encapsulation of objects. This means that each component can be changed without creating issues to other components. Furthermore, MVC provides decopuling of its components, which means that developers are able to work in parallel on different components of the pattern without interfering with each other.

Reverse Proxy Design Pattern

A simple proxy acts as an interface to refer an object in another machine. The reverse proxy offers a single point of access (with HTTP) to multiple Clients who want to access to several application servers. In practice, it is a wrapper for an object behind the scenes. Proxies also provides extra functionalities such as data caching (g.e. store the newest statistics available) and security (g.e. client-side firewall).

Motivations

It is useful to organize the requests of multiple clients and to save frequent requested informations. It also protects the application servers from external attack and slightly improve the overall performances.