



# POLITECNICO MILANO 1863

## Software Engineering 2 Implementation and Test deliverable Document SafeStreets

Version 1.0 - 12/01/2020

*Authors:*

Andrea Falanti

Andrea Huang

*Professor:*

Prof. Elisabetta Di Nitto

# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction</b>                                   | <b>4</b>  |
| 1.1      | Purpose and Scope . . . . .                           | 4         |
| 1.2      | Definitions, Acronyms, Abbreviations . . . . .        | 4         |
| 1.3      | Revision history . . . . .                            | 6         |
| 1.4      | Reference Documents . . . . .                         | 6         |
| 1.5      | Document Structure . . . . .                          | 6         |
| <b>2</b> | <b>Implemented functionalities</b>                    | <b>8</b>  |
| 2.1      | Introduction . . . . .                                | 8         |
| 2.2      | Functionalities included in the prototype . . . . .   | 8         |
| 2.3      | Functionalities excluded from the prototype . . . . . | 11        |
| <b>3</b> | <b>Adopted development frameworks</b>                 | <b>12</b> |
| 3.1      | Server . . . . .                                      | 12        |
| 3.1.1    | Programming languages . . . . .                       | 12        |
| 3.1.2    | Middleware . . . . .                                  | 12        |
| 3.1.3    | Relevant Packages . . . . .                           | 13        |
| 3.2      | Client . . . . .                                      | 14        |
| 3.2.1    | Programming languages . . . . .                       | 14        |
| 3.2.2    | Relevant Packages . . . . .                           | 14        |
| 3.2.3    | API . . . . .   | 15        |
| <b>4</b> | <b>Source code structure</b>                          | <b>16</b> |
| 4.1      | Server . . . . .                                      | 16        |
| 4.2      | Client . . . . .                                      | 17        |
| <b>5</b> | <b>Testing methodology</b>                            | <b>20</b> |

|          |                                  |           |
|----------|----------------------------------|-----------|
| 5.1      | Server . . . . .                 | 20        |
| 5.2      | Client . . . . .                 | 22        |
| <b>6</b> | <b>Installation instructions</b> | <b>23</b> |
| 6.1      | Server . . . . .                 | 23        |
| 6.1.1    | Prerequisites . . . . .          | 23        |
| 6.1.2    | Setup . . . . .                  | 23        |
| 6.1.3    | Testing and Analysis . . . . .   | 24        |
| 6.1.4    | Postman Setup . . . . .          | 25        |
| 6.2      | Client . . . . .                 | 25        |
| 6.2.1    | Prerequisites . . . . .          | 25        |
| 6.2.2    | Setup . . . . .                  | 25        |
| <b>7</b> | <b>Effort Spent</b>              | <b>27</b> |
| <b>8</b> | <b>References</b>                | <b>28</b> |

# Introduction

## 1.1 Purpose and Scope

The scope of this document is to provide an overview on how the SafeStreets developed prototype has been made, which technologies have been chosen for its implementation and how testing has been performed. It is also discussed how the prototype matches the requirements described in the "Requirement Analysis and Verification Document" and how the architecture follows the structure defined in the "Design Document".

It's important to say that the prototype is meant to represent a showcase of the system to the actual stakeholders, but it doesn't represent the final version of the software to be. The main focus of the prototype development is in fact to illustrate all functionalities of the service (either in complete or simplified versions), so strict functional testing has been performed on the prototype, while performances have been taken into account but not accurately tested.

## 1.2 Definitions, Acronyms, Abbreviations

### Definitions

- **Citizen or common user:** a common citizen, without any public office, who uses the application to report traffic violations.
- **Authorities:** the public officials who certify the violations reported on the application.
- **Municipality:** the public institution that provides traffic violation data to the application, and may consult and analyze intervention suggestions from it.

- **Operator:** a person working for SafeStreets who is responsible for signing contracts with authorities and municipalities.
- **Violation:** an event that does not conform with the traffic laws and that can be reported to authorities.
- **Intervention:** a possible public intervention produced by the application aimed at solving frequent violations in some areas of the city.
- **(Report) Supervisor:** authority that have validated or invalidated a report.

## Acronyms

- GPS = Global Positioning System
- S2B = Software to Be
- RASD = Requirement Analysis and Verification Document
- PEC = Posta Elettronica Certificata
- API = Application Programming Interface
- AJAX = Asynchronous JavaScript and XML
- GDPR = General Data Protection Regulation
- OCR = Optical Character Recognition
- DD = Design Document
- CPU = Central Processing Unit
- DBMS = DataBase Management system
- REST = Representational State Transfer
- SQL = Structured Query Language
- IO = Input Output
- UI = User Interface
- HA = High Availability

## Abbreviations

- $R_n$  = n-th requirement (defined in RASD).

## 1.3 Revision history

- Initial version 1.0

## 1.4 Reference Documents

- Specification document: “SafeStreets Mandatory Project Assignment”
- IEEE Std 830-1998 IEEE Recommended Practice for Software Requirements Specifications
- RASD v1.2
- DD v1.1

## 1.5 Document Structure

The Implementation and Test deliverable Document is composed by eight chapters:

**Introduction:** first chapter, contains a brief description of the document and its scope, along with definitions useful to understand the document content.

**Implemented functionalities:** second chapter, describes the requirements/functions that are actually implemented in the prototype (with motivations for including them and excluding others).

**Adopted development frameworks:** third chapter, contains info about the chosen programming languages and middleware, describing also the motivations of these choices.

**Source code structure:** fourth chapter, describes briefly the project structure of both client and server.

**Testing methodology:** fifth chapter, describes how the testing has been performed, the outcome and its usefulness.

**Installation instructions:** sixth chapter, provides the prerequisites and installation steps of the software.

**Effort spent:** seventh chapter of the document, shows how much effort each member of the group has spent on the various chapters of the document and the implemented prototype.

**References:** final chapter, contains links to material, information or documentation related to the content discussed in the document.

# Implemented functionalities

## 2.1 Introduction

In this section are listed all functionalities discussed in the RASD and DD document, also explaining how they are actually implemented and illustrating eventual relaxations that need to be refined for the final version of the software and also other eventual improvements. Anyway, the prototype covers at least a good part of all the functionalities required by SafeStreets system, providing a showcase of all the services offered by the system and also a very good starting point for implementing the final release.

Because the design document section 4 (Requirement traceability) provided already a map between the requirements and the components of the system, the following sections will focus directly on the server components, explaining how the functionalities are reached.

All the following functionalities are used by the client to request and display the correct data given by the server. Each client-side functionality is implemented with its screen, i.e. login, sign up, report a violation, data analysis and report visualization.

## 2.2 Functionalities included in the prototype

For API endpoints specification, please refer to the .yaml file inside 'Server/other/api' folder, it provides the whole API structure following the OpenAPI3 standard conventions.

- **Router:** implemented using *oas-tools* middleware, adhering to OpenAPI3 specifications for API standardization. *Oas-tools* reads and parse the .yaml file



containing the API specifications and links the endpoints of the API to the respective handlers in the controller files, successfully providing request and response handling of the server. Response code are coherent to the HTTP status codes, so that the client can easily understand if something went wrong with the request or if it was successful.

- **LoginManager:** completely implemented, when the login data is valid, the server generates a session, storing the user's id, the account type and the city of the organization (in case of authorities). The clients receive the cookies to set in the header of the response, so that they can easily save them and insert them in the headers of successive request to identify themselves and access functionalities restricted by authorization. Other services can check the presence of user session and perform checks using the data contained in it.
- **SignUpManager:** implemented with some relaxations compared to RASD scenario's specifications. The component successfully allows the registration of both authorities and citizens to the service, performing checks that the email format is valid, the password has at least 6 characters and that the domain of the email is coherent to the account type. This last check means that the domain in authorities registrations must be the same of one of the registered organization, while for citizens it's true the vice versa. Authorities accounts are automatically associated to their organization by the domain check, satisfying the RASD and DD specifications.
- **OrganizationRegistrationManager:** implemented as explained in DD, adding also the possibility of adding a new city in the database, in case it's not present when registering the organization, so that the registration of the organization can be completed successfully. A prototype of the SafeStreets operator interface is accessible through browser to the URL `'/pages/login.html'`, `'/pages/organization.html'`, `'/pages/city.html'` and can be used successfully only by logged account associated to an organization with type `'system'`. This type of organization can be inserted only by the system maintainer, performing the insertion directly into the database. Note that this interface needs actually major refinements, but can provide a simple showcase on how the interface scripting behaviour can be done.
- **ReportReceiverManager:** as described in the DD, allows the reception of a

form containing the data of the violation from the client, completing it with the id of the submitter, the timestamp of the server when the report is received, and the place and city of the violation using reverse geocoding service API. *Nominatim*, OpenStreetMap service API, has been used to get the place and city of the location provided in the report. For license plate recognition, clients can send a single photo to the apposite endpoint and the server will interrogate *platerecognizer.com* OCR service to provide the license plates to clients. Note that as the free plan of the OCR service has been used for prototype, only one request per second can be made.

- **ReportVisualizationManager:** two endpoint are provided by the server, one is used to receive a list of reports based on the account type, the other for accessing a single report by id. Both endpoint checks who made the request by accessing the user session, then provide only the reports accessible by them. In case of the endpoint of report for id, it means that if a report is not accessible by the user, an error is returned directly. Citizens can't check the authorities who supervised the reports, as a privacy measure.
- **ReportValidationManager:** a single endpoint accessible only by authority accounts simply allows to change the status of a report. A preliminary check make sure that the report is accessible by the authority (same city of their organization), then launch a SQL update command on the report tuple with that id, changing status and supervisor id (obtained from session).
- **DataAnalysisManager:** as explained in the DD, the components allows the filtering of the reports by date, city and violation type; this can be done by appending specific query parameters to the endpoint URL. Data structure returned differs based on the account type that made the request. Another endpoint is also provided to fetch all cities registered to the system, so that a client can find easily the desired city id for completing the data analysis requests (for example, clients can use a dropdown to show all the cities and then use the id as the value to send).

## 2.3 Functionalities excluded from the prototype

- In RASD use cases is stated that after the registration form is submitted, the user needs to confirm its email before he can actually use the account. In the prototype this step is omitted, because it's more focused on the actual functionalities of the system and the extra step of email confirmation would have required extra development time and extra testing time (one different email domain for organization) for an unnecessary function for a prototype showcase. Anyway, this step is essential in the final product to avoid fake registrations, especially fake authority registration that would compromise the privacy measures of the system.
- The prototype runs locally and it's not deployed, therefore the prototype should be adapted to run with a load balancer through multiple instances to maximize the performances.
- Even if not stated in DD, the passwords should be hashed and salted in final version of the product, because saving them in plaintext is not a good practice for security reasons. This could be done quite easily with the npm package *bcrypt*, but the step has been omitted for an easier debugging and faster insertion of test data, as the database is reseeded after each step the hashing and salting could probably affect testing time.
- The SafeStreets operator interface is actually accessible by everyone, but it only works with proper admin accounts. In production version these pages should be accessible only by authenticated account, to improve security of the system. It's layout and style should also be revamped and improved.

# Adopted development frameworks

## 3.1 Server

### 3.1.1 Programming languages

As programming language, JavaScript in a Node.js environment has been used for the server code. As explained in the "Other design decisions" of the design document, asynchronous programming has been chosen over a multi-thread approach as it provides better performances when handling a big amount of simultaneous requests. Node.js is built on the asynchronous I/O paradigm and it's pretty popular nowadays for its performances and scalability, which is one of the most relevant point of why it has been chosen for the development. One drawback of an asynchronous system is that each asynchronous operation must be coordinated with the other operations with care and also tested accurately to avoid unexpected behaviours, but the benefits in performances when avoiding synchronous operation and so not blocking the event loop are really important for high performance servers.

### 3.1.2 Middleware

- **OpenAPI(oas-tools):** the server stub has been generated from Swagger editor online, after the .yaml file with the specification has been written, following the rules of openAPI v3. The presence of the specification file provide a good documentation to the API, defining in a standardized way all the endpoint along with the consumed body and all the possible responses. The server contains the *oas-tools* middleware, that provides the routing functionality following the .yaml file, checking also that the received data correspond to the specified one, increasing the robustness of the server.

- **Express:** one of the most popular npm packages, it is used and customized along with the *oas-tools* package to handle request and responses. *Multer* and *Body-parser* packages are used to handle correctly the files and inserting the parsed data in the request body so that the *oas-tools* package can correctly see them without throwing an error (We think this is a bug of *oas-tools*, as the new openAPI specification are pretty recent and no official middleware has still emerged, customizing *multer* even if already present in the middleware actually resolved the problem).
- **Cookie-session:** allows to save sessions in the server and distributing the required cookies to clients, so that they can be recognized by the server after a successful login. Authentication is an important topic of our software as many features are restricted based on account type, sessions provide a good approach to recognize client and associating them with some data, so that also the following server computations can be faster as account type and city organization are stored directly in the session and don't need to be retrieved every time.

### 3.1.3 Relevant Packages

- **Knex:** package responsible for establishing the connection with the database and performing operations on the db.
- **Node-fetch:** used to generate requests to other servers, so that our server can call external services' API and await their responses.
- **Jest:** our chosen test framework. Provide a quite easy syntax, fast execution and many features for assertions.
- **Supertest:** used to generate request to the server itself and simulate a client for actually testing the API calls in an automatic way. Used along with jest test suites, it constitutes the core of the integration testing of the server.
- **Puppeteer:** testing API for running an instance of chromium (headless or not) and simulating browser interaction, for example fill-in forms. Useful to test the interface of web pages and the JavaScript code running in the browser.

## API

Server API endpoints follows the description defined in .yaml file, based on the high-level interfaces described in section 2.5 of the design document (Component interfaces). The controllers are associated to the Router component of the design document, while the actual interfaces described in the DD are the ones of the services, which respect quite closely the parameters defined in the architecture design, with the main difference in the types all ID parameters, that are actually integers and not strings. It's also necessary to state that the API endpoint for registering a city is missing from the DD diagram, but it's necessary because when registering an organization its city could be not present (if any report of violations in that city has been made). Also, in some handler requiring userID, the account type is also added as a parameter or replace the ID itself, because to avoid useless operation and improve the performance of the system also the type and account's organization city have been inserted in the session.

## 3.2 Client

### 3.2.1 Programming languages

The programming language chosen to implement the client application is Dart using the Flutter framework to create natively compiled cross-platform applications from a single codebase. This framework is able to provide a faster development process using its Stateful Hot Reload feature and its rich set of fully-customizable widgets.

### 3.2.2 Relevant Packages

Dart has a rich set of useful packages which can be found at <https://pub.dev> and can be easily imported using the Pub package manager. For this project the following packages were used:

- **dio**: powerful Http client used for raw and FormData POST requests.
- **cookie\_jar** and **dio cookie manager**: these two packages combined together are able to manage a client-side cookie session.
- **camera**: allows access to device cameras, used to display live camera and save

captured images to a file.

- **path\_provider** and **path**: a plugin for finding commonly used locations on the filesystem, used to set the documents path where the cookies are stored and set the temporary files path for photos when reporting a violation.
- **carousel\_slider**, **dropdownfield**, **flutter\_datetime\_picker**: packages used respectively for a carousel to show the images of a report, a dropdown field to allow the selection of a predefined set of items, and a date picker for the filters selection.
- **geolocator**, **flutter\_map**, **latlong**: packages used respectively for access to the GPS of the citizen when reporting a violation, map rendering using a OpenStreetsMap Tile server (<https://s.tile.openstreetmap.org/z/x/y.png>) and latitude-longitude object for the flutter map plugin.
- **charts\_flutter**: this package provides data visualization charts for data analysis, used for visualization of a time series bar chart to display the number of reports by day, a piechart for reports grouped by violation type and a piechart for reports grouped by city.

### 3.2.3 API

The client application has a single module (`rest_datasource.dart`) that provides all the methods to call the API endpoints of the server using the network module (`network_dio.dart`).

# Source code structure

## 4.1 Server

This section will highlight the main files and folders that compose the server, explaining the criteria of their subdivision:

### Files

- **index.js:** entry point of the server, when command **npm start** is executed the code contained in this file setups and start the server.
- **knexfile.js:** knex package configurations, so that the correct database is connected based on the node environment variable.
- **package.json:** contains all npm package dependencies required for running the server, that are automatically installed on the machine running the server when command **npm start** is performed. It also contains the scripts associated to npm commands.

### Folders

- **other:** contains all the code running on the server
  - **api:** contains the API specification file, parsed at server start by *oas-tools* middleware.
  - **controller:** contains source files associated to each server component. Each controller has functions that map an API endpoint, called by openAPI middleware after the input has been verified. Controller functions performs additional checks (especially on session fields to check what type



of account made the request), then if all its ok they call the services' functions. Controllers along with *oas-tools* provides the routing functionality.

- **service:** each source file contained in this folder constitute a server main component, described in section 2.2 of design document. Services' functions are called by controllers and perform the operations required to fulfill SafeStreets functionalities. The services communicate often with the database server, but to decouple the database connection and query construction from the modules, an intermediate service called *DataLayer* provide this functionalities to all services, avoiding also code duplication between the components.
- **test\_data:** contains the JSON files used to populate the server with test data and some images used in tests (file uploads).
- **utils:** contains source file that provide common utilities to other components.
- **public:** contains html pages and files accessible from the clients.
  - **assets:** contains css and js files associated to html pages.
  - **pages:** contains the html pages (excluding the homepage).
  - **reports:** contains a folder for each report stored in the database. Each report folder contains the photos associated to them.
- **tests:** contains unit tests and integration tests for all the server source files.
- **tools:** contains source files used by *knex* to create the database structure (migrations) and to populate the database after its creation (seeds).

## 4.2 Client

This section will highlight the main files and folders that compose the client, explaining the criteria of their subdivision:

### Files

- **pubsec.yaml:** file containing metadata about the used dependendecies and assets.

- **lib/main.dart**: contains the root widget of the application.
- **lib/config.dart**: configuration file defining the HOST address and the API endpoint's base url.
- **lib/routes.dart**: defines names for the main routes(screens) of the application, it will be passed to the main file to set the accessible screens.

## Main folders

- **lib**: contains the source code of the client
  - **data**: contains the class responsible of calling the correct server's API endpoint.
  - **models**: provides models for user, report, city objects; used to easily display information about each object.
  - **screens**: contains source files for each kind of screen, the implementation follows a Model-View-Presenter pattern so each screen is divided in a view class and a presenter class. The view class responds passively on the result of a service call of the presenter. The presenter class works as middleware between a view and the server, it uses the methods from the data module to call a specific service to obtain data for the UI rendering. The following folders group screens by the functionality they implement:
    - **analyze\_data**: contains a filter screen, map screen and a charts screen; the filter screen is used to retrieve data to be displayed on the map screen and charts screen.
    - **home**: contains a citizen home screen and an authority home screen to let the user choose which service to use.
    - **login**: the initial screen of the app, it allows the user to log in with his account or register a new account.
    - **photo**: contains a screen for taking a photo with the camera and a screen to check the taken photo and decide whether to save it or not.
    - **report\_violations**: contains the screen to report a violation after taking a photo and checking its license plate.

- **reports:** contains a screen displaying a list of available photos depending on the user and a screen containing more details of a single report, which is displayed when a report is clicked from the list.
- **signup:** contains a screen form citizen registration and one for authority registration.
- **utils:** contains a util file for enum class handling in dart and a network module.
- **widgets:** contains a customized carousel widget used to display images from a local folder or images from the network.
- **assets:** contains the logo of the app.
- **android:** contains the code generated for android platform; it contains the "build.gradle" file for build configuration and "AndroidManifest.xml" file which was used to set the permission requests to access to camera and location of the device.
- **ios:** contains the code generated for ios platform; the file "Runner/Info.plist" was used to set the permission requests to access to camera and location of the device.

# Testing methodology

## 5.1 Server

Proper unit testing and integration testing has been performed at the completion of each server component to make sure that all the new functions were working fine before proceeding with the remaining ones. As explained in the design document, a bottom-up approach has been used, building the system incrementally from the modules without dependencies and avoiding the necessity of stubs during testing.

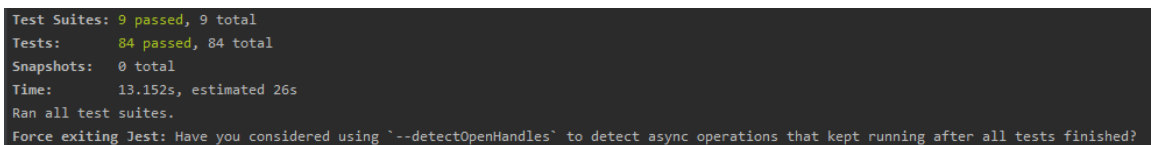
Server testing is performed on a separate environment from the one used for development and production, so that their databases can remain untouched. This can be done by setting `NODE_ENV = test` (automatically done with **npm test** command), so that the test database is used. *Jest* has been used as test framework to create and run the test suites. For each test file, before any of its test suites run, the database is completely destroyed and recreated from scratch, also, after each performed test, the database is reset to initial testing values by a *knex* reseed; this is done to maintain the testing environment consistent in all tests, without performing ad hoc reverting operations for each test, that is time expensive and also could lead easily to bugs. Even if *jest* supports parallel testing, sequential testing is forced by the test script (-i flag) because it's impossible to run multiple reseeds and migrations in the same time, so multiple databases would be required to perform parallel testing, specifically one for each test file.

Unit tests are focused mainly on the database interactions, making sure that the queries and SQL commands are working correctly; this has been done by making assertions on the returned objects of the queries and by checking that the records in the database are correctly modified when a insert or update command is performed.

Integration tests instead focuses on simulating API calls. API calls testing could

be done manually using Postman or the client, but an automatic approach is much more efficient because, even if coding all test suites has been time consuming, it's incredibly useful to check that all API endpoints handler are working fine even after further modifications, by just running a command. To simulate a client, *supertest* package has been used to send request to the server itself, then checking fields on the response received, like the http code and message, using usual *jest* assertions. Also the prototype of web interface for SafeStreets' operators is tested automatically thanks to *puppeteer* package, that is able to run an headless version of chromium to actually simulate interactions with a browser. Unfortunately, I couldn't find a way to insert the coverage of the JavaScript files running in the browser and so are excluded from Sonar report.

Here are the results of **npm test** on my desktop, as you can see all tests are correctly passed.



```
Test Suites: 9 passed, 9 total
Tests:      84 passed, 84 total
Snapshots:  0 total
Time:       13.152s, estimated 26s
Ran all test suites.
Force exiting Jest: Have you considered using `--detectOpenHandles` to detect async operations that kept running after all tests finished?
```

Figure 5.1: Test results

It's important to state that the test script also generate a coverage report, that can also be integrated in a *sonarQube* analysis to check the overall code quality of the project. Below is reported the result of the sonar analysis on the whole server folder, that is an important indicator of the quality of code delivered.

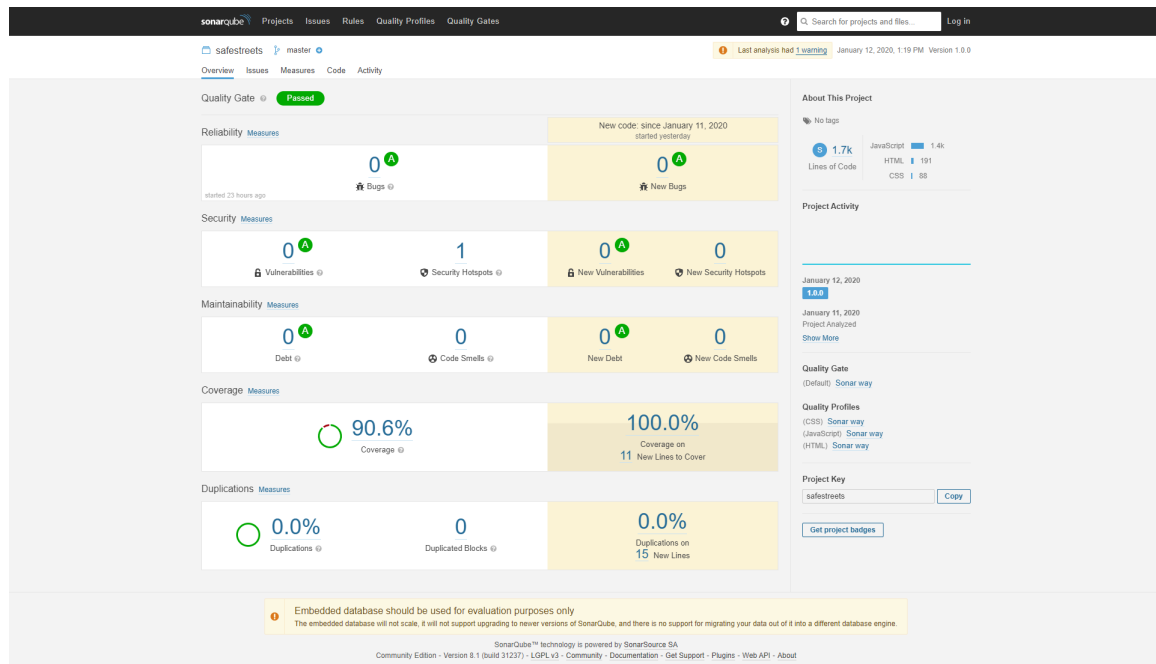


Figure 5.2: Sonar analysis results

## 5.2 Client

The testing on the client consists of unit tests on the business logic of the application. In particular, unit tests are performed for the encoding of json objects coming from the server (users, reports, cities) and data manipulation for representation on the charts of the data analysis screen. In the latter case, unit tests are performed to check whether the different kinds of reports grouping are correct.

# Installation instructions

## 6.1 Server

### 6.1.1 Prerequisites

- Node.js (v12.13.1)
- PostgreSQL (v12.1)
- Postman (v7.14.0)
- SonarQube (v8.1.0.31237) [Optional]

Make sure to install npm along node.js and pgAdmin during PostgreSQL if you want to avoid to use the terminal commands. If you want to use the PostgreSQL CLI, make sure "INSTALLATION\_PATH/PostgreSQL/12/bin" is inserted into paths system variable.

Note: official website's links are provided in references section.

### 6.1.2 Setup

1. Install all the required software
2. Clone the repository
3. Create two databases using pgAdmin (located in "PostgreSQL/12/pgAdmin 4/bin/pgAdmin4.exe") or the PostgreSQL CLI. With pgAdmin, wait for the browser interface to show up and insert the password inserted during installation, then right-click on databases to create database (name it as you want, for example "safestreets" and "safestreets\_test")

4. Go into "Server" folder of the cloned repository and add a .env file (without name), containing these fields:  
NODE\_ENV=development  
DB\_USER=*your postgresQL username*  
DB\_PASSWORD=*your postgresQL password*  
DB\_NAME\_DEV=*name of first database*  
DB\_NAME\_TEST=*name of second database*
5. Open the terminal and position yourself into "Server" folder.
6. Install knex globally with with npm, using command **npm install -g knex**. This is needed to seed the database later.
7. Start the server with command: **npm start**, the first time it's executed will require a bit of time because all required npm packages will be installed in repository's folder and the database structure will be generated.
8. After the server is successfully started, run this command in "Server" folder: **knex seed:run --env development** to populate the development database with data provided in json files located in "Server/other/test\_data". You can run the command any time to reset the database to these values.

### 6.1.3 Testing and Analysis

After performing all setup tests:

- In "Server" folder, while the server is running, use command: **npm test** to run the tests and generate a coverage report in "Server/coverage" folder. Server needs to be running because admin interface tests require the server to be on to open the pages, it's possible to execute the other tests when server is not running but that suite will fail, so be aware of that.
- While the server is running you can use Postman or the client to send requests (see also postman setup below). Unfortunately, SwaggerDoc page is not working for unknown reasons, most probably a bug of *oas-tools* middleware, because by copy-pasting the .yaml file content in SwaggerEditor online all is working properly.



- If you have installed sonarQube, you can run an analysis by installing globally the package with this command: **npm install -g sonarqube-scanner**, then while sonarQube is running its possible to generate a report by using the command **sonar-scanner** in the "Server" folder. To have the coverage set in sonar report, it's important that the tests have been already ran so that the coverage file is available.

### 6.1.4 Postman Setup

1. Click import button, select paste raw text and paste all the content of "Server/other/api/swagger.yaml", this will import the API into postman creating the collection.
2. Edit the collection to use http instead of https
3. Edit the two POST operations inside "report" to use in body a file for "photo" and "photo\_files" keys. Just put your cursor on the key name and a dropdown will appear on the right of the cell. If correct, you will see a "select file" button in the value cell.
4. Change body values of all requests with the wanted ones before executing.

## 6.2 Client

### 6.2.1 Prerequisites

- Flutter must be installed, for any issue run: **flutter doctor -v**.
- For iOS, an updated version of Xcode must be installed.

### 6.2.2 Setup

- If on Android Studio or IntelliJ open the Run menu and choose "Edit configurations ...", then add **—enable-software-rendering** to the Additional arguments field.
- Go to pubsec.yaml and run "Packages get" or run **flutter pub get** from the terminal.

- Before loading the application on a device, be sure to check the IP address of the server and update the configuration file in lib folder.
- Run the application.

# Effort Spent

Andrea Falanti:

| Document section               | Hours |
|--------------------------------|-------|
| Introduction                   | 1.25  |
| Implemented functionalities    | 2     |
| Adopted development frameworks | 1.25  |
| Source code structure          | 1     |
| Testing methodology            | 1.5   |
| Installation instructions      | 1.25  |
| Implementation                 | Hours |
| Server                         | 55.5  |
| Client                         | 0     |
| Total                          | 63.75 |

Andrea Huang:

| Document section               | Hours |
|--------------------------------|-------|
| Introduction                   | 0     |
| Implemented functionalities    | 0     |
| Adopted development frameworks | 0.75  |
| Source code structure          | 0.75  |
| Testing methodology            | 0.25  |
| Installation instructions      | 0.25  |
| Implementation                 | Hours |
| Server                         | 0     |
| Client                         | 52.5  |
| Total                          | 54.5  |

# References

- Node.js website:  
<https://nodejs.org/en/>.
- Node Package Manager (npm) website:  
<https://www.npmjs.com/>.
- PostgreSQL website:  
<https://www.postgresql.org/>.
- Postman website:  
<https://www.getpostman.com/>.
- SonarQube website:  
<https://www.sonarqube.org/>.
- Jest website:  
<https://jestjs.io/>.
- About OpenAPI3:  
[https://en.wikipedia.org/wiki/OpenAPI\\_Specification](https://en.wikipedia.org/wiki/OpenAPI_Specification).
- Swagger website:  
<https://swagger.io/>.
- Flutter website:  
<https://flutter.dev/>.
- Dart packages website:  
<https://pub.dev/>.
- OpenStreetMap website:  
<https://www.openstreetmap.org/>.