2021

POLITECNICO
MILANO 1863

Design Document-RODI

SE4G – SOFTWARE ENGINEERING FOR GEOINFORMATICS

AMIEVA, AUSTONI, SONDOQAH

Contents

1. Introduction

1.1 Design Document

One of the key elements in any software engineering culture is the use of defining software designs through design docs. These are relatively informal documents that the primary author or authors of a software system or application create before they embark on the coding project. Design document is a **representation** of a software design that is to be used for recording design information, addressing various design concerns, and communicating that information to the design's stakeholders.

Moving to the design document after finishing with requirements analysis document RASD, is a fundamental step that will guide the project developers and team members along the subsequent development path of a piece of software, this will include the implementation, integration and testing of system components.

IEEE defines software design documentation as "a description of software created to facilitate analysis, planning, implementation, and decision-making". In essence, a software design document (SDD) explains how a software product or a feature will be built to meet a set of technical requirements. If the requirements document describes the "**what**" of your project, the design document focuses on the "**how**".

Therefore, what will be described, explained, and covered is the following:

-The software structure in terms of the functions and/or classes/packages/modules

-Use Cases mapping in the Web App and the implementation mechanism

-Project Database: As one of the main parts of any software or application; the structure of our project database will be defined. The **schema** of each table, the attributes **data type**, and the whole **database configuration** will be explained in detail.

1.2 Final Product Description

As it was explained in the RASD document, this Web App is a proposed version of **Roads Network Maintenance System** which is an application can be used in the first case as a starting point for the authorities in the Ministry of Transportation and the responsible bodies in Milano to organize and

facilitate the process of fixing the **Pavement Distresses** in the streets which is a widespread phenomenon in Milano. In other words, this Web App can be considered as an enhanced database for the Pavement Distresses and streets anomalies for the responsible authorities in Milano.

What we will be offering in the final product is a Web App based on a database collected manually by individuals from different areas in Milano using the corresponding Mobile App of Epicollect5 platform, later this collected data will be offered in our Web App in such a way can be reached by the workers in the Ministry of Transportation where they can use this data as a starting point for any operational approach. The provided data and the offered information will be supported by a set of analytical and well-describing features to make the data more meaningful and indicative for the user.

The Web App accessibility is divided into two main streams, the first mainstream will be as declared before for the **specialized users** where the use here will be for operational purposes. The second one will be dedicated for **Citizen users** who are expected to use the Web App for individual purposes, one possible scenario is when a citizen user uses the Web App to check the presence of the distresses at a certain area so he can avoid taking this way while using his bicycle heading for his destination.

Finally, one of the main provided services of our Web App is submitting a maintenance request where a registered citizen user can submit a request asking for maintenance for a certain distress available in the Web App database.

2. Software Structure

The **software architecture** of a system depicts the system's organization or structure, and provides an explanation of how it behaves. A system represents the collection of components that accomplish a specific function or set of functions. In other words, the software architecture provides a sturdy foundation on which software can be built.
The architecture of a software system is a metaphor, analogous to the architecture of a building. It functions as a blueprint for the system and the developing project, laying out the tasks necessary to be executed by the design teams.

There are multiple high-level architecture patterns and principles commonly used in modern systems. These are often referred to as architectural styles. The architecture of a software system is rarely limited to a single architectural style. Instead, a combination of styles often make up the complete system.

In our case, our Web App will be implemented following the **Three-tier architecture;** which is a well-established software application architecture that organizes applications into three logical and physical computing tiers: the presentation tier, or user interface (**Web Server**); the application tier (**Application Server**), where data is processed; and the data tier (**Database Server**), where the data associated with the application is stored and managed.

In this section of our design document we will try to define the component of each tier and project this definition into our Web App.

2.1 The Web Server

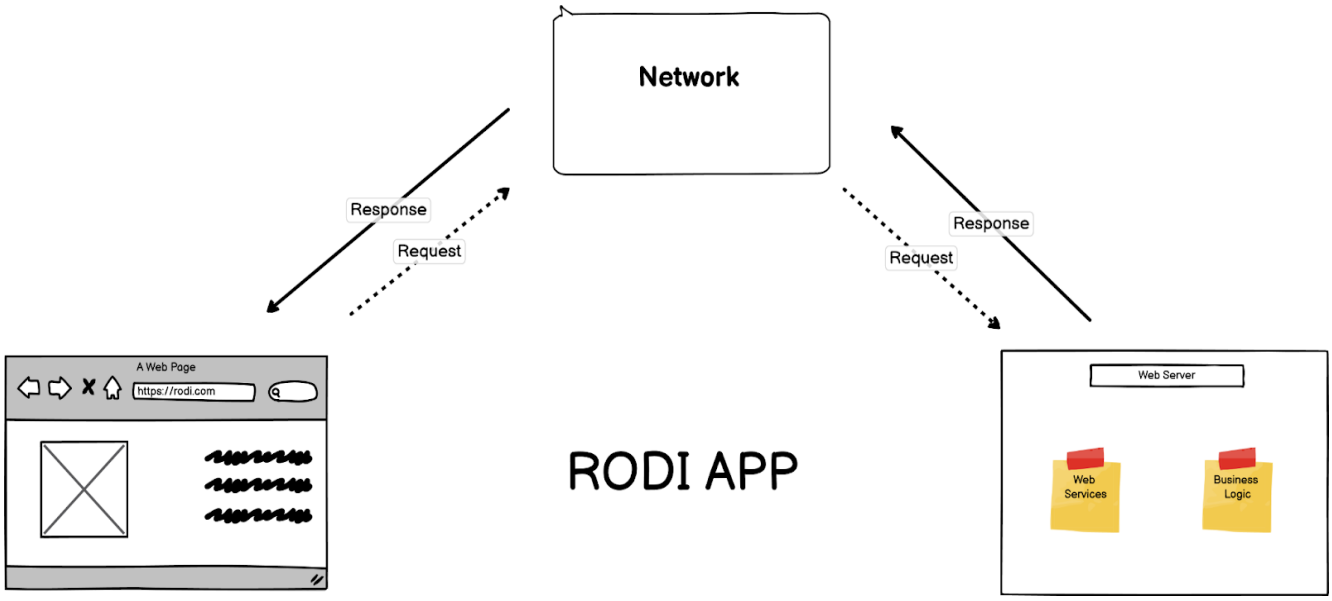
The web server is the presentation tier and provides the user interface. This is usually a web page or web site, such as an ecommerce site where the user adds products to the shopping cart, adds payment details or creates an account. The content can be static or dynamic, and is usually developed using HTML, CSS and JavaScript.

First, you type in the URL and as you hit Enter, your browser prepares to recognize this URL, because it needs to know the address of the server where the page is located (**we are working on a local host in our case**). So here a HTTP request will be sent. The browser here sends the request to the found IP address using the HTTPS protocol.

Second, the web server processes the request. The web server where RODI APP is located catches the request and sends it to the storage area to locate the page and all data that follows with it. But its route is held via Business Logic (also called Domain Logic and Application Logic). BL manages how each piece of data is accessed and determines this workflow specifically for each application. As BL processes the request, it sends it to storage to locate the looked-for data.

Third, you receive your data. Your response travels back to you and you see the content of the web page on your display. The graphical interface you see when scrolling RODI or any other website is called the front end of an application – it depicts all UX and UI components so that a user can access the information they came looking for.

Here we have a simplified chart for the process flow:



2.2 Application Server

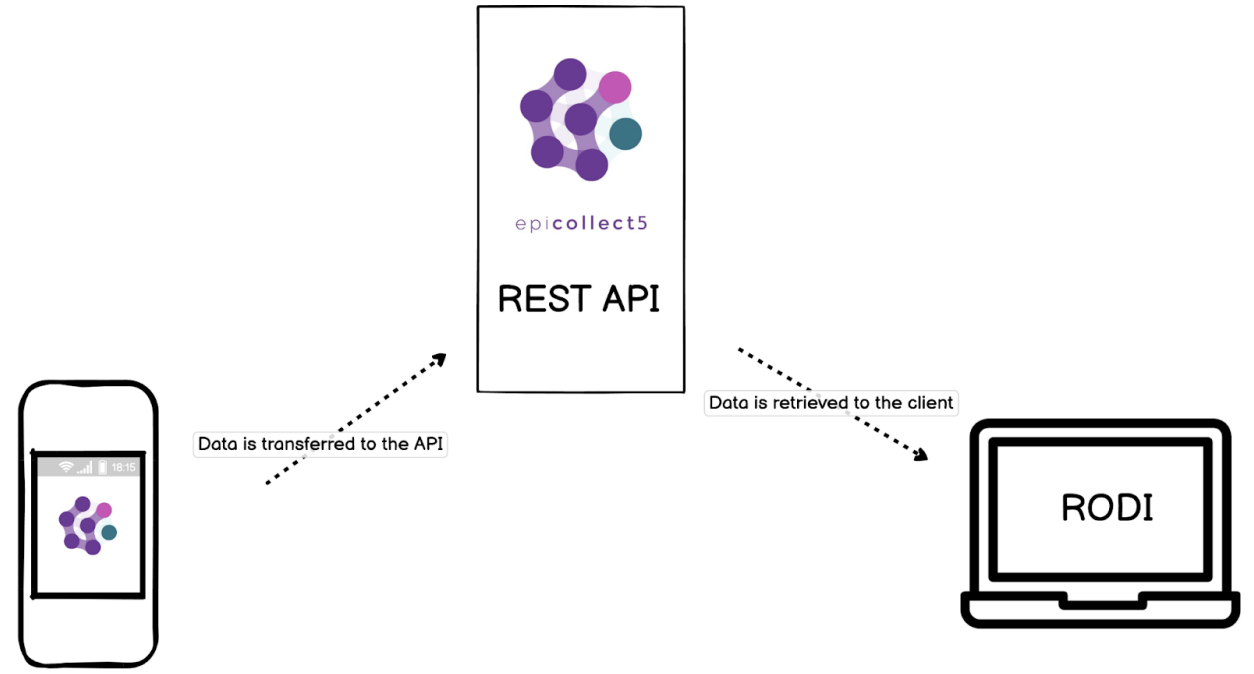
The application server corresponds to the middle tier, housing the business logic used to process user inputs. To continue with our Web App flow, this is the tier that queries the inventory database to return the distress distribution, or submit maintenance requests. This layer is often developed using Python, Ruby or PHP and runs a framework such as e Django, Rails, Symphony or ASP.NET, for example, In our Web App it will be written using python, and hence we'll be dealing with a WSGI server.
In this section; we will introduce the breakdown of our Web App internal structure of the logical functions.

2.2.1 Data retrieval from EpiCollect5 REST API

A REST API is an application programming interface (API or web API) that conforms to the constraints of REST architectural style and allows for interaction with RESTful web services. REST stands for representational state transfer and was created by computer scientist Roy Fielding.

An API is a set of definitions and protocols for building and integrating application software. It's sometimes referred to as a contract between an information provider and an information user—establishing the content required from the consumer (the call) and the content required by the producer (the response). For example, the API design for a weather service could specify that the user supplies a zip code and that the producer replies with a 2-part answer, the first being the high temperature, and the second being the low.

In other words, if you want to interact with a computer or system to retrieve information or perform a function, an API helps you communicate what you want to that system so it can understand and fulfill the request.
What we will have in our project is exactly the same, the used API will be Epicollect5, which is a mobile & web application for free and easy data collection. It provides both the web and mobile applications for the generation of forms (questionnaires) and freely hosted project websites for data collection.
What makes our project more complete in comparison to typical Web Apps is that it tried to satisfy all the application specifications cycle in addition to filling the gaps which exist in the real world, so it's a kind of whole system introduced to solve the problem we defined for the scope of this project.
The whole process can be simplified into this graph below:



You can think of an API as a mediator between the users or clients and the resources or web services they want to get. It's also a way for an organization to share resources and information while maintaining security, control, and authentication—determining who gets access to what.
And since the dataset was created by us especially for this project; the form was built in a way where just the appropriate attributes were set by the form builder provided by Epicollect5, **so we need no clean step for the data retrieved here.**

Finally, the used libraries and modules can be summarized as follows in this table below:

Used Libraries	**Requests, JSON**
Inputs	-The input of this functionality will be this url: 'https://five.epicollect.net/api/export/entries/MRNM?per_page=100'
Outputs	The response of this request will be converted into text before converting into JSON file.

2.2.2 Preprocessing of the data

After retrieving the data using the API and converting it into JSON format; the data has to pass through a preprocessing step. Once that data is in JSON format it should be transformed to a better format where

it is easy to handle, and here comes the role of panda. Panda is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language. However, the fact that we are dealing with geospatial data requires us to handle the data in different ways since the geospatial data have some properties different from those for the regular data (not ionic etc.). The good news is that python is well prepared for such situations, GeoPandas is an open source project to make working with geospatial data in python easier. GeoPandas extends the datatypes used by pandas to allow spatial operations on geometric types. Later the data will have the format of geodata frame where it will be ready for any kind of spatial operation.
The table below breaks down the structure of this logical functionality:

Used Libraries	**Pandas,Geopandas**
Inputs	-The input here will be at first the JSON format of the data, then after being transformed into dataframe, this dataframe will be the second input.
Outputs	We'll have the dataframes and the geodata frames as outputs from this functionality.

2.2.3 Web App - Database Interaction

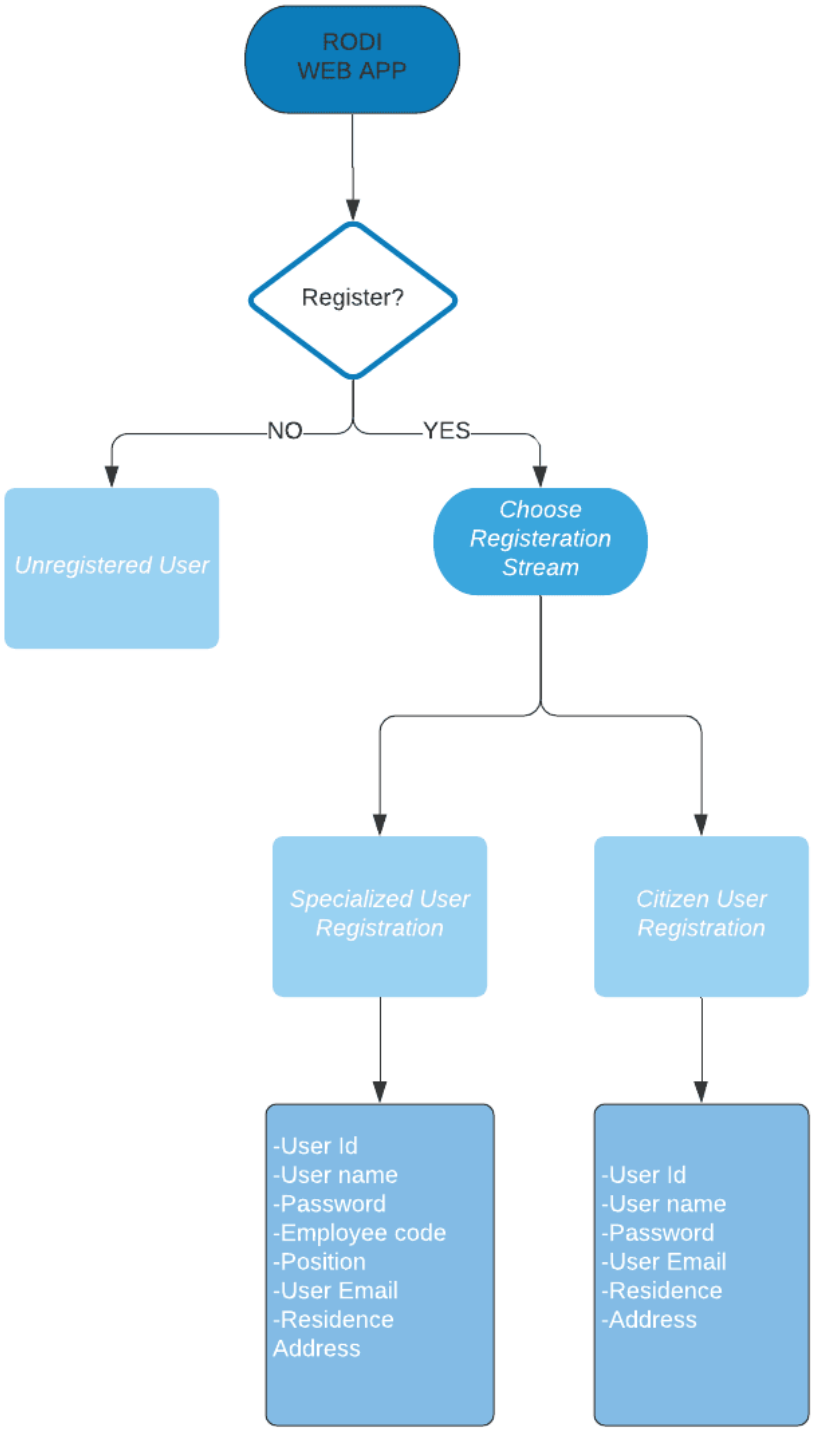
As for any Software following the design conventions, an interaction between the business logic and the database is needed, The primary role of a database is to store and display updated information in a web application. The database will be created once the python code runs for the first time, and then the tables can be found in the chosen database in the DBMS. The created tables will vary based on the defined schema in the SQL commands, which is basically a database language but will be activated here by using a set of python modules which are SQLAlchemy, Psycopg2. Also, the fact that we are dealing with geospatial data makes it necessary to import some specialized libraries such as GeoAlchemy2. All the the mentioned libraries and more will be used to create the set of tables in our database, the structure and the schema of each table varies based on the use and the purpose of the table, but basically we will have tables to manage the web app accessibility for different types of users, and table to handle the maintenance request submission. Basically the used libraries and modules in the web app database interaction can be concluded as following:

Used Libraries	**Pandas, GeoPandas, SQLAlchemy, GeoAlchemy2, Psycopg2**
Inputs	Here we will have two categories of data: -The data related to the users such as (username, password, email,etc.) -The data related to the geospatial dataset which is retrieved from the epicollcet5
Outputs	PostgreSQL tables with different data types and attributes.

2.2.4 Users Registration and Access control

Web applications need access controls to allow users (with varying privileges) to use the application. They also need administrators to manage the application's access control rules and the granting of permissions or entitlements to users and other entities. Various access control design methodologies are available. After the creation of the database, a set of tables related to the Web App users will be ready to be filled with the users information. Here we will have two streams for the registration process; one for the normal citizen user and one for the specialized use.
The implementation logic will be coded in python, and SQL commands will be written in order to complete the process.
Splitting the registration process in such a way makes it possible to control the access of each category of users, and we'll be using this feature to give the specialized more accessibility over the content and the functionalities in the Web App, the differences in accessibility will be clarified later.
Once the user decides to register; he will be asked to choose one of the two registration streams, and then he will be asked to provide the necessary information to successfully complete the registration process, the filled information will be saved in the database in two different tables for the citizen and specialized users.
After finishing the registration, the user will be redirected to the login page to fill his credentials in order to sign in, the entered credentials will be compared to those in the database, if they are correct the user will successfully log in, otherwise an error message will appear asking to re-enter correct ones. However, the specialized user will not be able to log in until his registration request is approved, this will be done when the super admin checks his request manually (real world phenomena) and then accepts his request using an approach that will be discussed later.

The flow chart below shows the breakdown of this functionality:



2.2.5 Geospatial Visualization of the data

As one of the best descriptive methods, Data visualization gives the user us a clear idea of what the information means by giving it visual context through maps or graphs. And since we're dealing with geospatial data, visualizing the data over a map will identify the data distribution in the area of interest which is Milano in our case. A wide set of base maps can be used to accomplish

this approach, in addition to some specialized libraries which support the use of interactive maps. In RODI Web App; we'll be using the Folium library to render built-in maps; OSM was rendered as a basemap for the main visualization of the distribution of the entities in the field. The used libraries and modules for successfully handle this function can be listed as follows:

Used Libraries	**GeoPandas, Folium**
Inputs	Here the entities of our project will be the input for this functionality, the visualization will be carried out based on the coordinates of each entity.
Outputs	As an output we'll have a visualization of the dataset entities over the basemap.

2.2.6 Geo-Statistical and parametric graphs

We agreed that we need data visualization because a visual summary of information makes it easier to identify patterns and trends than looking through thousands of rows on a spreadsheet. It's the way the human brain works. Since the purpose of data analysis is to gain insights, data is much more valuable when it is visualized. Therefore, we will be including a wide set of statistical charts to better explain and identify any valuable insights. The good news that python has many powerful modules to support such functionality, also there is some interaction between JavaScript and python to benefit from extra modules like Plotly. The components of this functionality can described as follows:

Used Libraries	**Plotly, Matplotlib, Geopandas, Numpy**
Inputs	The inputs of this functionality will be essentially based on the data retrieved by the API and stored in the Database. However, Using some filtering approaches offer more manipulated parametric input.
Outputs	As an output we'll have a visualization of the specified charts-will be determined later-. It's worth mentioning that the output of this functionality will be just dedicated for the Specialized users since this is the purpose of the web app in the first place as we discussed before.

2.2.7 Template Engine

2.2.7.1 Flask application

Flask is a lightweight WSGI web application framework. It is designed to make getting started quick and easy, with the ability to scale up to complex applications. It began as a simple wrapper around Werkzeug and Jinja and has become one of the most popular Python web application frameworks.
Flask offers suggestions, but doesn't enforce any dependencies or project layout. It is up to the developer to choose the tools and libraries they want to use. There are many extensions provided by the community that make adding new functionality easy.
A directory will be created by Flask to establish the connection between the database and the web server. This directory will be divided into two main parts. This directory will be splitted into two parts; one will contain the python codes for generating the functions and architecture of the connection, and one allocated for the templates. The first part will be discussed here:

-The first half is python code that contains the functions for interacting with the database. This approach allows the user to send the specific request. These requests are the main key for establishing the connection between the user and the database to retrieve information from the database and present them on the client browser in the html format.
-The second half of the python code is for defining the architecture of the database connection and user client.
The main function imported from Flask will be explained briefly in this section:

1-Render_template()

render_template is a Flask function from the flask. templating package. render_template is used to generate output from a template file based on the Jinja2 engine that is found in the application's templates folder. Note that render_template is typically imported directly from the flask package instead of from flask.

2-Request.form()

The key/value pairs in the body, from a HTML post form, or JavaScript request that isn't JSON encoded, Assign to the data requested from a specific field in the Web App.

3-Redirect

When called, it returns a response object and redirects the user to another target location with specific status code.

4-url_for()

url_for in Flask is used for creating a URL to prevent the overhead of having to change URLs throughout an application (including in templates). Without url_for, if there is a change in the root URL of your app then you have to change it in every page where the link is present

2.2.7.2 JINJA Engine

Jinja is a fast, expressive, extensible templating engine. Special placeholders in the template allow writing code similar to Python syntax. Then the template is passed data to render the final document. The templates we have will be an empty skeleton, they basically contain variables and expressions which will be replaced with their values. JINJA template engine which
dynamically builds HTML pages using familiar Python concepts such as variables, loops, lists, and etc. based on logic and context.
In general, there are two main html files for generating the web request and displaying retrieved data from database:

1-**base.html** : This template defines a simple HTML skeleton document that you might use for a simple two-column page. It's the job of "child" templates to fill the empty blocks with content.

2-**index.html** : As any Web Application we will have other components which will build the structure of the App, those components require html code to to render them. Those components can be described as children for the parent (base.html), and checking the parent file you'll find free blocks to be filled by the child templates.
These are basic codes for generating the web application through client users(browser). Based on the user cases and the functional requirement on the RASD document we have different components, partials and sub pages. We will define templates(child files) based on the different functions and user cases. In the table below mentioned the most important tags and expression for defining templates (JINJA elements) :

Tag	Description
{% block %}	Contains statements
{% block title %} {% endblock %}	It's a placeholder for a title, it's used to customize the title for each table in our Web App.
{{ url_for('index')}}	The function will return the url of the desired page then it will be rendered.
{% block content %} {% endblock %}	A content identified by the child template will replace this block.

Html Elements

Tag	Description
<html> … </html>	The root element
<head> … </head>	The document head
<title> … </title>	The document title
<body> … </body>	The document body
…	A section heading
…	A paragraph
<a> … 	A link
…	A block-level container for content
<img>	An image

CSS Usage

Cascading Style Sheets (CSS) is a style sheet language used for describing the presentation of a document written in a markup language such as HTML. CSS is a cornerstone technology of the World Wide Web, alongside HTML and JavaScript.

CSS is designed to enable the separation of presentation and content, including layout, colors, and fonts. This separation can improve content accessibility, provide more flexibility and control in the specification of presentation characteristics, enable multiple web pages to share formatting by specifying the relevant CSS in a separate .css file which reduces complexity and repetition in the structural content as well as enabling the .css file to be cached to improve the page load speed between the pages that share the file and its formatting.

Application Server Integration

After describing the majority of the App server components, now it's time to conclude the interaction between each of those components to create the complete image of the App server and its working mechanism.

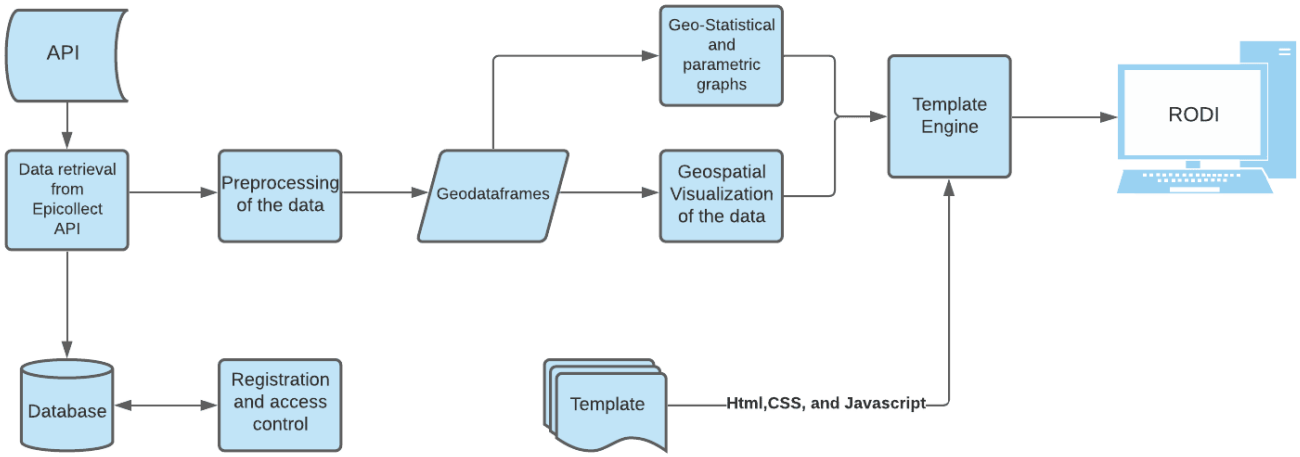


Figure 4: Interaction between different components of the RODI Web App

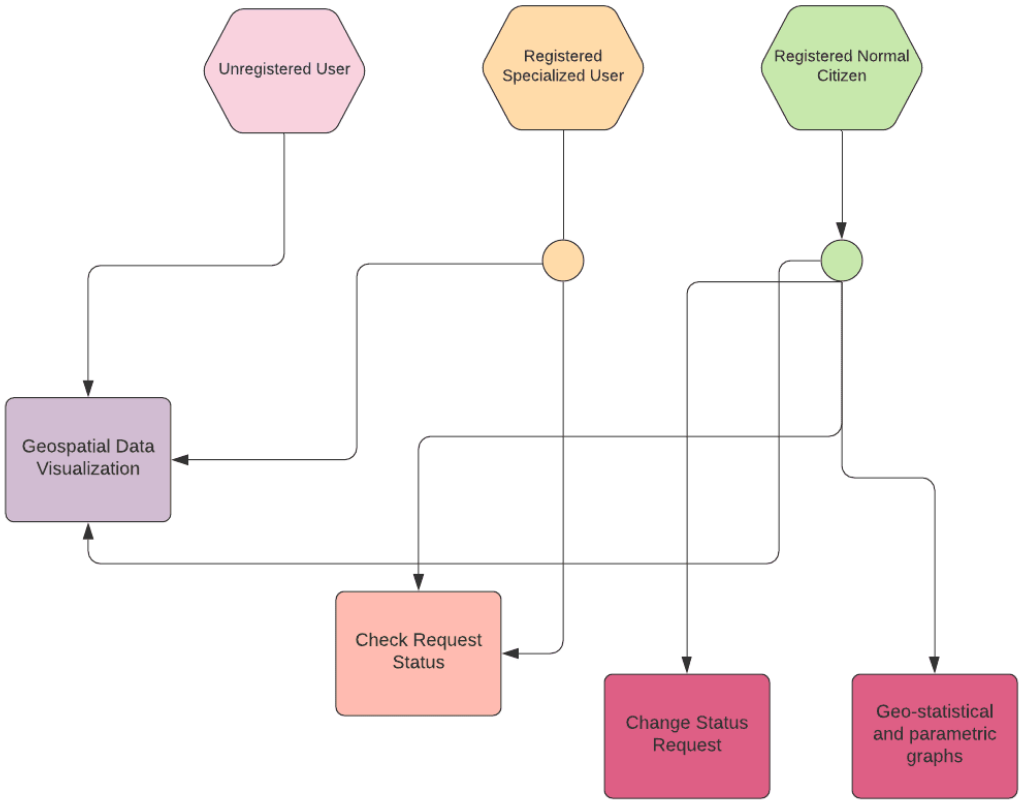


Figure 5: User's categories interaction with RODI components

2.3 Database Server

The database server is the data or backend tier of a web application. It runs on database management software, such as MySQL, Oracle, DB2 or PostgreSQL, for example.
Here we'll be using PostgreSQL DBMS to handle this tier in our Web App.
The Structure of the database will be defined later, here we'll try to identify the main used functions:

Function	Description
Connect()	The call of this function will create a new database connection to the PostgreSQL database server.
Fetch()	To fetch data from the database to the Python main programme.
Commit()	The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.
Close()	To close the connection with the database.

3. Use Cases

UC1: Member Registration:

Defining the user's categories and controlling the web app accessibility cannot be done without accomplishing such steps. Since the registration process is splitted into two streams; one for the citizen users, and other for the specialized user, it was necessary to find the appropriate logic to implement it. What we did is initializing a column in Users table to indicate the user type -more information will be introduced later in the Database design section-.
This use case is well described in the software components **Users Registration and Access control**.
The procedure can be described in the following structure:
1- The user opens the homepage for the Web App
2- The user chooses specialized user registration or normal registration, he will have two different pages for each type of registration.
3- Based on the user's choice; he has to fill the required information, those filled information will be sent to different tables in the database.
4- In case the user chooses the normal registration stream, the registration will take place and he will be able to log in anytime if the entered credentials match those in the related database table.
5- In case he chooses to register as a specialized user, his request to register will be directed to the **super admin**, the super admin will have web page to accept and reject each registration request based on the entered information manually.

UC2: Submitting a maintenance request.

This use case will be done by using the mobile app for Epicollect5, the user will be provided with guiding instruction to do this in the Web App.
Once the entity is collected, this will be considered as a **request submission**.

UC4: Updating maintenance requests

As referred in the accessibility control section, this use case is offered only for the specialized user, the specialized user will have the ability to change the request status once the maintenance process is performed. This use case will interact with the database in such a way we'll have a column in the Data_table for which is called status, different values will be assigned to each status to facilitate the code logic flow. The specialized user will have a GUI to change the status of those requests, once this change is performed the values of the status field in the database will be changed.
More details about the mechanism will be added later.

UC4:Updating of maintenance requests

As referred in the accessibility control section, this use case is offered only for the specialized user, the specialized user will have the ability to change the request status once the maintenance process is performed. This use case will interact with the database in such a way we'll have a column in the Data_table for which is called status, different values will be assigned to each status to facilitate the code logic flow. The specialized user will have a GUI to change the status of those requests, once this change is performed the values of the status field in the database will be changed.
More details about the mechanism will be added later.

UC6:Filtering and Visualization of Charts

This use case as described before will be offered only for the registered specialized users, the charts were generated using Plotly, where the map used here is OSM base map, the filtering is done by using logical statements based on the filtering schema. Also some Machine learning models are used for analytical purposes, those models are imported using Sickit learning library in Python.

UC7: View Map

This is a use case where it can be reached by all the categories of users, here a base map generated by Folium is used to offer a geospatial visualization for the entities over the basemap.
The entities are retrieved from the database to appear over the basemap and popup will appear once the pointer passes over the entity.

UC9: Contact Us

A separate web page will be detected for this use case where the user from all categories will be able to submit a contact request, the information filled will be redirected to a the Contact database table and an email will be sent to the request submitter to inform him the contact request was successfully sent and will be answered soon.

4. Database Design

The DBMS used to create the database of our web application was PostgreSQL. The schema of our database, the structure of our database in terms of the tables, and the relationships between them are defined in this section.

4.1 Database Schemas:

Our database consists of four tables:

1-The SYS_Table

The table contains information for the **registration** and **login** use cases, also in case of **submitting a maintenance request** use case.
The schema for this table will be as follows: (user ID, usernames, passwords, and residence_address,User_type).
The attributes has the following description:

Attribute	Description
User ID	This ID value is going to be generated automatically and it's going to be used as a primary key
Usernames	It's going to be inserted by the user during the registration and it will be used in the login use case
Password	Inserted by the user as well in the registration process and it's required for the login
Residence_address	The user is required to insert his address of residence in the registration
User_type	This attribute will be used to categorize the different registered users. Super Admin: the value 2 will be assigned to this attribute Admin: the value 1 will be assigned to this field Citizen User: the value 0 will be assigned
Employee_code	This attribute is just for the specialized user and it's going to be assigned in the registration step
Position	This attribute is only for the specialized user and it's going to be assigned in the registration process

2-The data_Table

The table contains all the data retrieved from the REST API, they will be transferred to the database to have better accessibility and perform the desired analysis over the data.
The schema for this table will be as follows: (Distress_ID, Time_Coll, Date_Coll,Coll_email,latitude,longitude,Pave_type,Distress_type,Distress_size,Risk_level,Status).

Attribute	Description
Distress_Id	Automatically generated ID for each distress and will be the primary key of the table
Time_Coll	The time at which the distress is observed
Date_Coll	The date of the day in which the distress is observed
Coll_email	The email of the distress observer
latitude	The latitude of the distress position
longitude	The longitude of the distress position
Pave_type	The type of the pavement in which the observed distress exists
Distress_type	The type of the observed distress within the suggested options
Distress_size	The avg size of the distress
Risk_level	The level of corresponding to the observed distress
Status	The status of the distress maintenance request

3-The Contact Table

This table will contain the information coming from the contact us page, once the user fills a contact request, the data in the form will be saved in his table.
The schema of the table will be (Contact_Id,Full_name,Contact_Reason,Email,Contact_text)

Attribute	Description
Contact_ID	The primary key of the table and will be generated automatically
Full_name	The Full name of the contact request submitter
Contact_reason	The reason of the contact request submission
Email	The email of the request submitter
Contact_text	The body text of the contact request

4.2 Tables Design:

1-The SYS_Table

Field Name	Datatype	Mandatory/Optional	Notes
User Id	Integer	Mandatory	Primary key
User name	VARCHAR(50)	Mandatory	Not Null
Password	VARCHAR(50)	Mandatory	Not Null
User Email	VARCHAR(50)	Mandatory	Unique,Not Null
User_type	Integer	Mandatory	Not Null
Residence Address	VARCHAR(100)	Optional	
Employee code	Integer	Optional	Unique
Position	VARCHAR(50)	Optional	

2-The Data_Table

Field Name	Datatype	Mandatory/Optional	Notes
Distress_ID	Integer	Mandatory	Primary key
Time_Coll	TIME(0)	Mandatory	Not Null
Date_Coll	DATE	Mandatory	Not Null
Coll_email	VARCHAR(50)	Mandatory	Unique,Not Null
latitude	DECIMAL(8,6)	Mandatory	Not Null
longitude	DECIMAL(9,6)	Mandatory	Not Null
Pave_type	VARCHAR(50)	Optional	
Distress_type	VARCHAR(50)	Optional	
Distress_size_avg	DECIMAL(4,2)	Optional	
Risk_level	VARCHAR(50)	Optional	
Status	VARCHAR(10)	Mandatory	Not Null

3-The Contact Table

Field Name	Datatype	Mandatory/Optional	Notes
Contact_ID	Integer	Mandatory	Primary Key
Full_Name	VARCHAR(50)	Mandatory	Not Null
Contact_reason	VARCHAR(50)	Mandatory	Not Null
Email	VARCHAR(50)	Mandatory	Not Null
Contact_text	VARCHAR(500)	Mandatory	Not Null

References

1- https://flask.palletsprojects.com
2- https://www.ibm.com
3- https://www.synopsys.com