POLITECNICO DI MILANO

M.Sc. Geoinformatics Engineering

Software Engineering for Geoinformatics

a.y. 2021-2022

Software Design and Test Plan Document

COCO
CUMBI

Authors:

Stefano Brazzoli	895477
Martina Giovanna Esposito	996431
Mattia Koren	993021
Gaia Vallarino	996164

May 25 2022
Version 1.0

Index

1 Introduction

Purpose

According to IEEE guidelines, the design document:

> *"... is an integral part of the software development process, and a vital source of information. Technical documentation is a means to make knowledge about a software system explicit that would otherwise only remain implicit knowledge in the heads of the development team that easily gets lost over time Further, documentation presents information at a higher level of abstraction than the system implementation, which makes it an important means for communication among stakeholders."*

In essence, the purpose of this document – which is none other than a technical description - is to provide the reader the general blueprints and a detailed overview of the structure of the project, by illustrating and establishing important and otherwise undiscussed characteristics that will serve as guidelines in the future of the software's development.

The general reader of this document might or might not already be an expert or knowledgeable enough to thoroughly follow the structural flow of this document; therefore, *Appendix A* is left for the more amateur public of this document.

We hereby define the intended audience of this document as:

- The Client: Public Administration of the Province of Pichincha, from here on now referred to as *PA* or *the client.*

- The Project Manager of the team, from here on now referred to as *PM.*

- The Developers and the Testing Team, referred to as *we/us*

- The End Users, i.e., the employees of the PA in question, from here on now referred to as *users*

The purpose of our project is to help the client, who is the PA of the province of Pichincha in Ecuador, make and implement policies supporting its many activities dedicated to the forestation and reforestation of its territory, through a desktop application.

The software will use data coming from a forest census of the Canton of Ruminahui, which represents trees as georeferenced points in the area of study.

This project wants to provide a way for PA to access, retrieve, visualize, and analyze data starting from a tree census in the area of the Ruminahui canton in the Pichincha province of Ecuador. More information about the purpose of the software can be found in the complementaryRAS document.

The main goal of this software is to help the regional government make analyses and decisions regarding environmental policies, such as calculating the biodiversity impact of a new construction, the level of endangerment of species within the canton or the best area for a reforestation effort or plan.

The software will be at first accessible only through computer technology. This was set as a priority during the preliminary meetings with the client, since it emerged that the web app will primarily be used within the government office spaces, where each employee is equipped with a personal computer device. We do not exclude that, at a later date, the device could be implemented by the team of developers for other devices, such as tablets and/or phones.

This document will provide a further insight into the architecture of the software COCO CUMBI and is intended for a complementary read with the latest version of the RASD document. The suggested way to approach these documents is to first read the RASD and only then begin the SDD, so that the reader can gain a better and more detailed understanding of the components of the software, together with its implementation, and test plan.

The RASD can be found on our GitHub repository at the following link:

github.com/gaiavallarino/SE4GI

Structure of the Software Design Document

This document is intended to be tightly linked with the RAS document of the software, in order to comply with the intended requirements established in the latter one.

The following chapters will characterize the following aspect of our software *Coco Cumbi*:

- Chapter 2: *Databases*. We will thoroughly explain the main characteristics of our database, given the importance of the database connection and usage for the software.
- Chapter 3: *Structure*. Here the structure is described in detail, together with libraries, functions and different implementation aids.
- Chapter 4: *Application of Use Cases and Test Plan*. Here the implementation of the use cases is described together with the testing strategy to be sure that they work as intended.
- Chapter 5: *Team Management*. A brief report on hours worked, deliverables, roles and responsibilities of the team.

2 Database

There are three main databases that come into play in the design and implementation of *Coco Cumbi*. The software will use and retrieve information from the following services:

- *EpiCollect5*, (from here on now also known as EC5) source of our tree census
- *PostgreSQL*, DBMS, intermediate step between EC5 and our web application.

The data will be retrieved through the use of REST API's. To better articulate the procedure, what happens is that the tree census data from EC5 are processed and later transferred to a PostgreSQL DB; this operation will happen at a regular interval of time. This process is needed because of the several advantages that using an intermediate database (i.e., PostgreSQL) instead of directly fetching data from EC5 brings. Among these, we have the possibility to leverage DBMS tools and abilities through the interface between our programming language, Python, and PostgreSQL itself. We also reduce the risk of losing data and we ensure the data availability. All these reasons and many more obviously improve the performance of our software.

EC5 Tree Census Dataset

We have stated before that the dataset we are taking in consideration comes from a tree census of our region present on EC5. Said data is structured in a table where each tree specimen (one per row) has different characteristics and is georeferenced in the territory through UTM coordinates. This will come in handy later on when we will need to render the two-dimensionality of the data on an auxiliary map, while the remaining characteristics provided in the table can be considered as our third-dimension metrics.

The dataset is named "*Censo Forestal del Canton Ruminahui*" and it consists of more than 27.000 georeferenced specimens found in the abovementioned Canton. The complete dataset can be found at the link below.

five.epicollect.net/project/censo-forestal-del-canton-ruminahui/data

The original EC5 project table, before our preliminary data rehash, appears as follows and can be found in the RAS document as well:

Parameters	Description
Numeric tree ID	Indicates the tree identifier, every ID is unique
Date	Date on which the corresponding tree was sampled.
Census Area ID	Indicates the area to which the tree belongs, every area has a unique ID
Group	Group of pertinence of the census agent who added the piece of data
Common Name	Name commonly used to refer to a specific type of tree
Scientific Name	Name used to define an organism which is unique to that organism and the same in any language
Tree status indicator	Indicator of the tree health status. Ex: Acceptable, Medium...
Coordinates	Latitude and Longitude
Written coordinates	UTM Northing and UTM Easting inserted by the census agent
DBH	Tree diameter at breast height
Height	Height of the specimen
Crown diameter	Diameter (in m) of the crown of the specimen
Crown radius	Radius (in cm) of the crown of the specimen

Sector	Name of the place where the data belong to, for instance the name of a city
Property	Private or Public
Risk	Risk factor associated with the tree location such as inclined stem

Table 1 – EpiCollect5, *Censo Forestal del Canton Ruminahui*

PostgreSQL – PostGIS DB

Our software will interact with a DBMS (*DataBase Management System*), which in this case will be the abovementioned PostgreSQL, for data storing and management while it is running on a WSGI server. One of the main advantages and reasons why we are using PostgreSQL is the extensibility of it, which allows us to exploit the extension PostGIS, which is a spatial database extender for PostgreSQL object-relational database. It adds support for geographic objects allowing location queries to be run in SQL.

Here, the data tables will contain primary keys, i.e., unique identifiers that will allow us to univocally identify an entity. Primary and non-primary keys grant the possibility of logical relationship among tables.

In our case we won't provide the user with an Entity-relationship diagram, since at this moment in time our software only entails two uncorrelated databases (trees and users)

After the rehashing of the table, where we only keep the data that we think will be useful for the analysis and visualization of data, we are left with the following table:

Parameters	Descriptions
treeID	*Object;* Indicates the tree identifier, every ID is unique
date	*Object;* Date on which the corresponding tree was sampled.
censusArea	*Object;* Indicates the area to which the tree belongs, every area has a unique ID
group	*Object;* Group of pertinence of the census agent who added the piece of data

commonName	*Object;* Name commonly used to refer to a specific type of tree
scientificName	*Object;* Name used to define an organism which is unique to that organism and the same in any language
status	*Object;* Indicator of the tree health status. Ex: Acceptable, Medium...
dbh	*Float, Tree diameter at breast height*
height	*Integer;* Height of the specimen
crownRadius	*Integer;* Radius (in cm) of the crown of the specimen
crownDiameter	*Float;* Diameter (in m) of the crown of the specimen
sector	*Object;* Name of the place where the data belong to, for instance the name of a city
property	*Object;* Private or Public
risk	*Object;* Risk factor associated with the tree location such as inclined stem
x	*Float;* corresponds to the Longitude
y	*Float;* corresponds to the Latitude

Table 2 – Coco Cumbi, *Trees*

In the preprocessing, some attributes were ignored since they were not going to be of use in our software. Attributes such as "written coordinates", which represented the coordinates inserted by the census agents with their own positioning devices. These coordinates might not be accurate to the level we need and might not be suitable for our calculations, therefore were ignored and later dropped from our tables.

For what concerns the cases of the user signing up, logging it or performing any other user-related operations, there is also a database table containing the user's information.

Attribute	Description
Email	VARCHAR(255) PRIMARY KEY contains the email of the user
Firstname	VARCHAR(255) NOT NULL contains the firstname of the user

Lastname	VARCHAR(255) NOT NULL contains the lastname of the user
Password	VARCHAR(255) NOT NULL contains the password chosen by the user

3 Structure

The web application presents both static, (i.e., information that can't be changed through user interaction, developed as HTML code) and dynamic (i.e., information that can be changed through user interaction, developed as Python code).

The software architecture is structured on a three-layer architecture, which is composed by an HTTP server, an application server, and a database server. It is also divided into two main programs, that concern the *database configuration*, which creates the database structure, and the *flask application*, which manages the python code and the Jinja template engine.

The *database configuration* is needed in order to have a server where the required data can be stored, retrieved, and from which it is possible to be manipulated at the user's need. The operations that are needed in order to interact with PostgreSQL can be summed in opening a connection, sending a request to execute a SQL command and/or change something; in the end, we need to close the connection to the server. The interaction between our software and PostgreSQL happens with the aid of the following libraries and functions:

- psycopg2, *connect*
- sqlalchemy, *create_ engine*

In our configuration script we write the code to allow for the retrieval and processing of the data that is to be stored in the DB. In this case we have used the following libraries:

- requests
- json
- pandas
- geopandas

When we have imported the dataset from EC5, we retrieved the data through a REST API by requesting it (request library); successively, the data need to be converted into a json file (json library). Once this is done, we can convert the data into a Pandas dataframe (pandas library) and eventually, turn it into a geodataframe (geopandas), by introducing the geometry attributes to the dataframe. In this case, since we are dealing with tree specimens, we have added the *point* type.

Once the processing is over, the data is imported into the PostgreSQL DB; In order to accomplish this, we need a connection to be set up to manage the link with PostgreSQL. Subsequently, the application will run once and a new table will be created in the DB, with the data correctly copied into the corresponding table. Moreover, when a new piece of data is added into the EC5, the PostgreSQL DB should automatically append it to the PostgreSQL DB table.

On the Flask application side of things, we start by saying that Flask is a web microframework. It is a Python module that lets one develop web applications with ease through its easily extendable core. It is a WSGI web app framework that allows us to do url routing and exploit a template engine (i.e., Jinja).

Compared to other frameworks, it is more simple, since it only has the tools necessary to support the interaction with the WSGI server. It includes both the Python side of the software and the templates. The application server provides the logical operations needed for the software requirements. The web browser then interacts with the web server and the interaction between WSGI and the web server is defined by WSGI specifications.

The *Python code* has to answer to the requests that the users give through the web server, through the WSGI interface, in order to perform the functions and retrieve the data needed.

A quick overview of the different libraries and functions used is given below.

- flask
- psycopg2
- werkzeug.security
- bokeh
- geopandas
- pandas
- shapely.geometry

After this, we need to create the application instance through flask. This is done by passing as arguments the name and template folder implemented for the app. We also set a secret key to some random bytes.

At this point, we can define the different functions that we have used to implement our software. They are divided into different categories, such as database connection, user profile interaction, queries, statistical analysis.

As a small note, the functions that need the user to make a request to the server need to have a declaration of *@app.route* to map the function with the html page.

get_dbConn()

If we are not yet connected to the DB. We pass the .txt file “dbconfigTest.txt”, which contains the database name, the username and the password. The function reads this file and saves the string; it then connects the DB.

close_dbConn()

If we are already connected to the DB, the function closes the connection.

splitcoordinates(df)

for every element in the writtenCoordinates column the function first locate the position of the division symbol(; or -,) and then split the element into two different elements (latitude and longitude) and convert them into float type, then store them into two lists and add this lists to the dataframe.

USED FUNCTIONS: replace(), find(), float(), len()

INPUT: dataframe (df)

OUTPUT: a new dataframe equal to the input dataframe with two more columns for latitude and longitude.

heightrange(max,min,df)

The function receives as input two values, a max and a min, if are both None it returns the input dataframe, if these aren't any None it makes a query and returns a new dataframe with only the elements that satisfy the conditions for max and min.

USED FUNCTIONS: query()

INPUT: max, min, dataframe (df)

OUTPUT: a new dataframe filtered with the max or min or each value for height attribute

crownrange(max,min,df)

The function receives as input two values, a max and a min, if both are None it returns the input dataframe, if these aren't any None it makes a query and returns a new dataframe with only the elements that satisfy the conditions for max and min.

USED FUNCTIONS: query()

INPUT: max, min, dataframe (df)

OUTPUT: a new dataframe filtered with the max or min or each values for crownrange attribute

interactive_map()

It allows the user to visualize the georeferenced data on the map

USED LIBRARIES: flask, bokeh

USED FUNCTIONS: GetPointCoords, splittedcoordinates

searchname(worser,df)

The function receives as input a search word, if it is None, the function returns the input dataframe, if it is not None, it makes a query and returns a new dataframe with only elements with the commonName attribute equal to the search word.

USED FUNCTIONS: query()

INPUT: a search word (worser) and a dataframe (df)

OUTPUT: a new dataframe with only elements with commonName attribute equal to the search word received as input

searcharea(worser,df)

The function receives as input a search word, if it is None, the function returns the input dataframe, if it is not None, it makes a query and returns a new dataframe with only elements with the area attribute equal to the search word.

USED FUNCTIONS: query()

INPUT: a search word (worser) and a dataframe (df)

OUTPUT: a new dataframe with only elements with area attribute equal to the search word received as input

searchgroup(worser,df)

The function receives as input a search word, if it is None, the function return the input dataframe, if it is not None, it makes a query and return a new dataframe with only elements with the group attribute equal to the search word.

USED FUNCTIONS: query()

INPUT: a search word (worser) and a dataframe (df)

OUTPUT: a new dataframe with only elements with group attribute equal to the search word received as input

searchsector(worser,df)

The function receives as input a searchword, if it is None, the function returns the input dataframe, if it is not None, it makes a query and returns a new dataframe with only elements with the sector attribute equal to the search word.

USED FUNCTIONS: query()

INPUT: a search word (worser) and a dataframe (df)

OUTPUT: a new dataframe with only elements with sector attribute equal to the search word received as input

register()

The function connects to the registration page, takes down the credentials submitted by user and checks for correctness and for the presence of an existing account with the submitted email; then if there are no errors it adds a new element in the pa_user table with the submitted credentials and returns the login function, otherwise it returns the error and the registration page.

USED LIBRARIES: flask, werkzeug.security, psycopg2

USED FUNCTIONS: request.form, find(), len(), get_dBConn(), cursor(), execute(), fetchone(), close(), redirect(), url_for(), render_template(), commit(), generate_password_hash(), session

INPUT: none

OUTPUT: login function, error html page or registration html page

login()

The function connects to the login page, takes down the credentials submitted by user (email and password), connects to the database and checks if there is in the pa_user table an element corresponding with the email, and then, if there is, checks if the password submitted is equal to the one stored in the database. If there are no errors it stores the email address in the session variable and returns the home function. If there are errors, it returns the error html page with the corresponding error message.

USED LIBRARIES: flask, werkzeug.security, psycopg2

USED FUNCTIONS: request.form, get_dBConn(), cursor(), execute(), fetchone(), close(), redirect(), url_for(), render_template(), commit(), check_password_hash(), clear(), session

INPUT: none

OUTPUT: home function, login html page, error html page

Home()

The function recall for the *load_logged_in_user()*, if it is True return the homepage html, if it is False redirect to the login function

USED LIBRARIES: Flask

USED FUNCTIONS: load_logged_user(),render_template(),redirect()

INPUT: None

OUTPUT: homepage html page or login function

load_logged_in_user()

The function takes the email value from the session variable, then checks if it is None, if it is true the g.user variable is set to None and returns False, if it is not None it connects to database and loads to the g.user variable the element corresponding to the email and returns True

USED LIBRARIES: Flask, psycopg2,

USED FUNCTIONS: get(), get_dBConn(), cursor(), execute(), fetchone(), close(), commit()

INPUT: None

OUTPUT: True or False

logout()

The function removes the values stored in the session variable and returns the home function

USED LIBRARIES: flask

USED FUNCTIONS: session,clear(), redirect(), url_for()

INPUT: none

OUTPUT: home function

Template Engine

On the template engine it is possible to generate templates and insert some constructs, such as loops, in order to modify and work with the output that will be modified. In the case of our software, we use Jinja as a template engine to be able to dynamically change, manipulate and visualize our HTML files. The template is a file that contains expressions and variables that will be replaced once the page is rendered with the actual dynamic values.

In our case we have not rendered a true *base.html* to work as a parent template, since our application doesn't have a main layout. This will be partly substitute by the other pages, with which it will be possible to navigate from one html file to the other.

- login.html: through this page, the users can login with their credentials
- signup.html: through this page, non-registered user can register on the web app providing their credentials and personal information
- main.html: this is the page that can be reached after having logged in; here the user can find the information about the software, perform queries and visualize the results of queries, and access to other html pages, such as visualize data, (ndr. in a later version) contacts, about us, etc.
- map.html: through this page the user can visualize on an interactive map the data
- contacts.html: page implemented mainly for homepage navigation testing. In the future implementation of the software, it will contain the team members' contacts.

The html pages used *CSS* (Cascading Style Sheets) as a style language to create the appearance of the web app.

Some of the pages have not been implemented yet, or are still WIP, thus they have not made the list on this version of the document, but they will be integrated into the following version of this document, during the implementation phase of the project.

4 Uses Cases and Test Plan

UC1: Registration

@app.route('/register', methods=('GET', 'POST'))

The function receives two methods:

- for POST the function has to:
  - get information from user (name, lastname, email, password and checkpassword) and save them in corresponding variables
  - check for the presence of all mandatory credentials, if not send error and return error page
  - check for a valid domain in email(in our case: mail.polimi.it), if not send error and return error page
  - check the length of password (it has to be at least 8 character long), if not send error and return error page
  - check that password and checkpassword have the same values, if not send error and return error page
  - check if there is already an existing account in the database: if there is send error and return error page, if not register the account in the database and redirect to login page

- For GET the function has to:
  - redirect user to signup page

Test Case 1 – Correctly signs up user

inputs
- User accesses registration page and inserts:
  - First Name: Marco
  - Last Name: Rossi
  - Email: marco.rossi@mail.polimi.it
  - Password: fff56sj2
  - Password confirmation: fff56sj2

hypothesis
- There is no corresponding email stored into the database

output
- The account is saved and stord into the database, registration worked correctly

Test Case 2 – Last name is missing

inputs
- User accesses registration page and inserts:
- First Name: Marco
- Last Name:
- Email: marco.rossi@mail.polimi.it
- Password: fff56sj2
- Password confirmation: fff56sj2

hypothesis
- Last Name is missing

output
- The user visualizes the message "Surname is required"

Test Case 3 – Wrong email domain

inputs
- User acesses registration page and inserts:
- First Name: Marco
- Last Name: Rossi
- Email: marco.rossi@gmail.com
- Password: fff56sj2
- Password confirmation: fff56sj2

hypothesis
- The email inserted has not the rights to registrate and then access the application

output
- The user visualizes the message "Domain not authorized"

Test Case 4 – Password doesn't respect the criteria

inputs

- User accesses registration page and inserts:
- First Name: Marco
- Last Name: Rossi
- Email: marco.rossi@mail.polimi.it
- Password: fff56
- Password confirmation: fff56

hypothesis

- The password is not long enough

output

- The user visualizes the message "Password needs to be at least 8 characters long"

Test Case 5 – Password and confirmation don't match

inputs

- User accesses registration page and inserts:
- First Name: Marco
- Last Name: Rossi
- Email: marco.rossi@mail.polimi.it
- Password: fff56sj2
- Password confirmation: ggg56sj2

hypothesis

- The password confirmation field doesn't correspond to the password field

output

- The user visualizes the message "Different password confirmation"

UC2: Log in

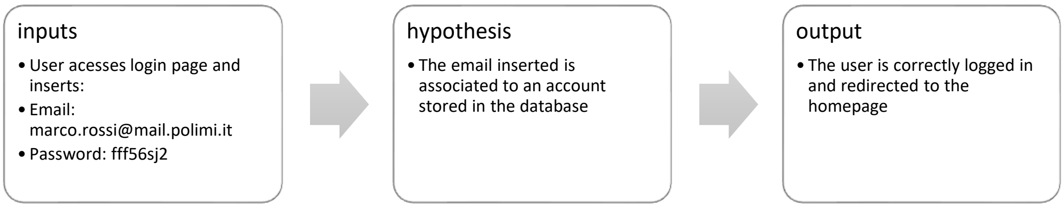
@app.route('/login', methods=('GET', 'POST'))

The function receives two methods:

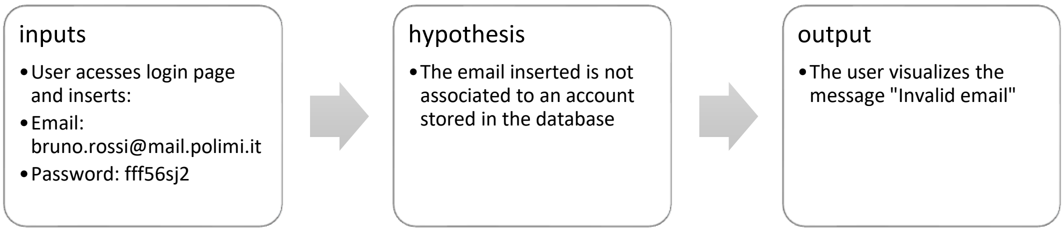
- for POST the function has to:
  - get information from user (email,password) and save them in correspondig variables
  - check for the presence of an account in the database corresponding with the email submitted by user if not send error and return error page, if yes check if the password submitted by user is equal to the password stored in the database
  - if there are not errors store the email in the session variable and redirect to the home page of the application

- For GET the function has to:
  - redirect user to html login page

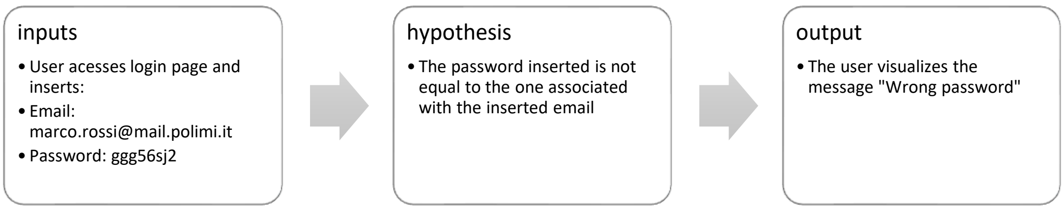
Test Case 1 – User correctly logs in



Test Case 2 – User is not registered



Test Case 3 – Wrong password

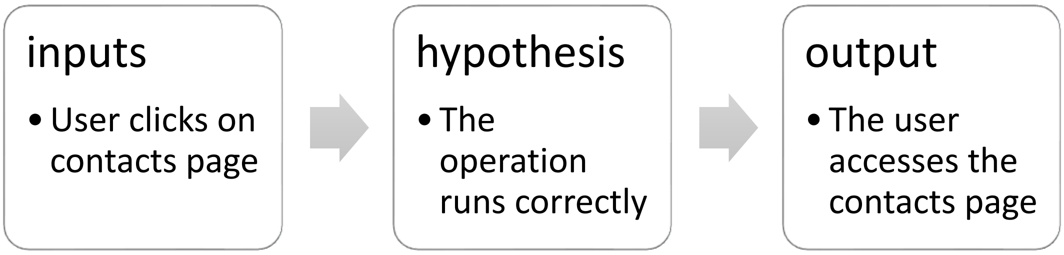


UC3: Homepage Navigation

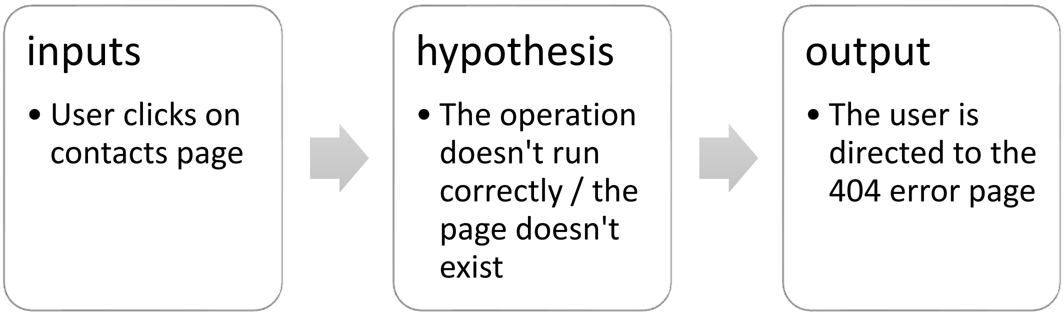
@app.route('/home', methods=('GET')

the function calls the function load_logged_in_user(), if it is True: it redirects user to the html homepage, if not it redirects the user to login.

Test Case 1 – Navigation in the page works



Test Case 2 – Navigation in the page doesn't work, redirect



UC4: Data Request/Query

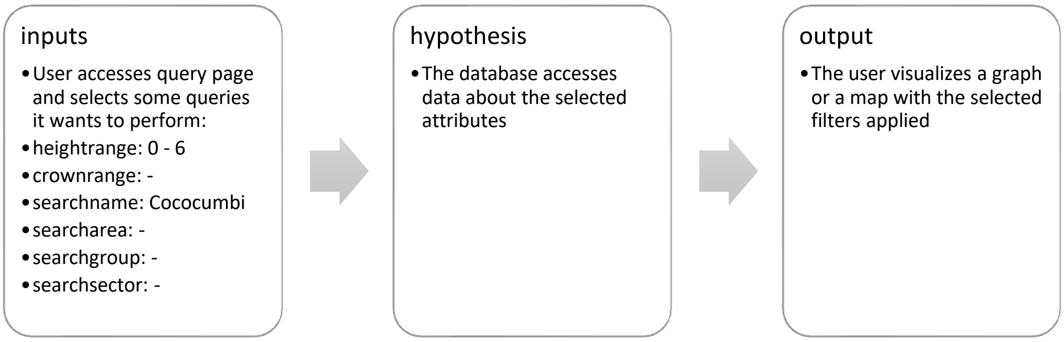
@app.route('/query', methods=('GET', 'POST'))

The function receives two methods:

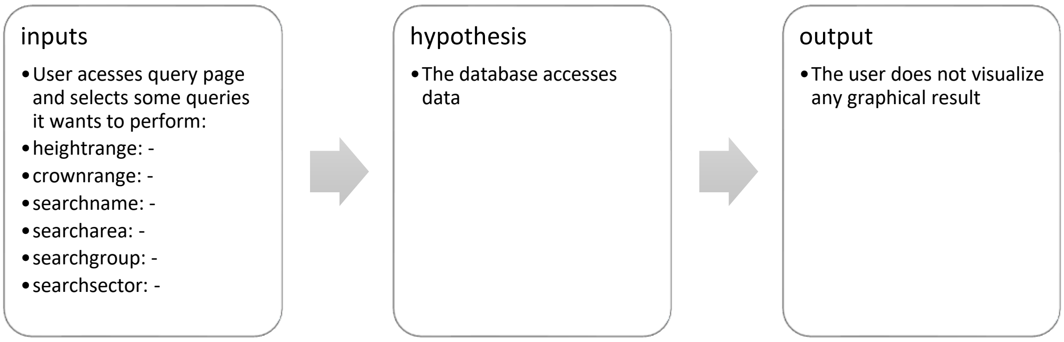
- for POST the function has to:
  - receive the filter values from the user (height, crown diameter, common name, group, area, sector) and save them in corresponding variables

- run in succession the query functions (heightrange, crownrange, searcharea, searchname, searchgroup, searchsector) and return query page with query results
- For GET the function has to:
  - return the query page

Test Case 1 – Querying correctly



Test Case 2 – Querying does not allow for data visualization

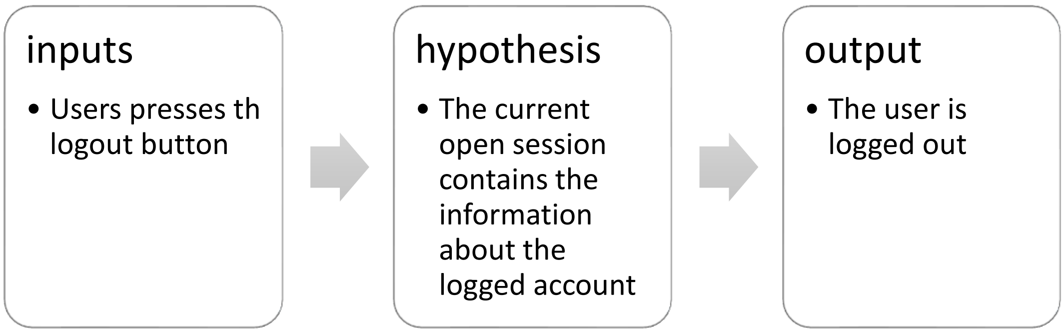


UC5: Log out

@app.route('/logout', methods=('GET')

the function removes the email from the session variable and redirect user to the homepage function.

Test Case 1 – User correctly logs out



UC7: Result Representation

Not yet implemented at the time of writing version 1.0

UC8: Interactive Map

Not yet implemented at the time of writing version 1.0

5 Team Management

The project workload for this document and the linked operations needed to write it, was equally split among the team members in term of time and amount of work.

Given the abovementioned deliverables, the roles and workload were split four-ways as follows:

- Brazzoli, *BE*: PostgreSQL, DB realization and connection
- Esposito, *BE*: query and data visualization
- Koren, *BE*: function creation, general backend coding, coordination and connection among files .
- Vallarino, *FE*: realization of HTML and CSS code, SDD report

The team would like to note that, even though the work was evenly split, working as a team entailed multidisciplinary roles that encompassed other team members' mansions as well as one's own, as we came together to help one another. For sake of simplicity, this is not reported in the above list, but let it be known that no single job can be fully attributed to a single team member.

Team Member:	Number of Hours Worked:
Stefano Brazzoli	25
Martina Giovanna Esposito	25
Mattia Koren	25
Gaia Vallarino	25
TOTALE	100

Appendix A – Commonly Used Terms

WSGI

The Web Server Gateway Interface (Web Server Gateway Interface, WSGI) has been used as a standard for Python web application development. WSGI is the specification of a common interface between web servers and web applications.

jinja2

jinja2 is a popular template engine for Python. A web template system combines a template with a specific data source to render a dynamic web page.

HTTP server

It is a computer program, or a software component included in another program that plays the role of a server in a client-server model, implementing the server part of the HTTP/HTTPS network protocols. An HTTP server waits for the incoming client requests (sent by user agents, like browsers, etc.) and for each request it answers by replying with requested information, including the sending of the requested web resource, or with an HTTP error message

Appendix B - Definitions, Acronyms and Abbreviations

Find here an updated list of commonly used and referenced acronyms and abbreviations found in the document; also find some definitions of terms that might not be common knowledge to some of the intended audiences of this document.

Terms	Descriptions
ID	Identifier
System	The application we are designing
DBH	Diameter at Breast Height
User	A person who utilizes a computer or network service
User authentication	A security process which ensures that a user cannot access another user's profile if not in possess of their credentials
Queries	A request for data or information from the Database
Georeferenced data	Data tied to a known Earth coordinate system
PA	Public Administration of the Province of Pichincha
PM DB	Project manager of the team
DBMS	Database Management System
EC5	Epicollect5
GPS	Global Positioning System
WIP	Work in Progress
BE	Back-end
FE	Front-end

Bibliography

Elisabetta Di Nitto, *Software Engineering for Geoinformatics – Slides*, 2022

Details & Update Log

DETAILS

DELIVERABLE	SDD
TITLE	Software Design & Test Plan Document
SOFTWARE NAME	Coco Cumbi
AUTHORS	Brazzoli S., Esposito M.G., Koren M., Vallarino G.
VERSION	1.0
DATE	May 25th, 2022
DOWNLOAD PAGE	github.com/gaiavallarino/SE4GI
COPYRIGHT	Copyright © 2022, Brazzoli S., Esposito M.G., Koren M., Vallarino G. – All rights reserved

UPDATE LOG

Version	Date	Description
1.0	May 25th, 2022	First draft and submission