L.M. Geoinformatics Engineering
AA 2019/2020
Software Engineering

POLITECNICO
MILANO 1863

Design document

Delivery: Design Document
Version: 1.0
Authors: Marianna Alghisi
Gabriele D'Ascoli
Martina Pasturensi
Chiara Pileggi

Revision History

Date	Version	Description	Author
10/05/20	<1.0>	Initial version of Document	Group 3
3/05/20	<2.0>	We made corrections, added the Test part and improved the use cases	Group 3

Table of contents

1. Introduction

Our project aims to show to the Users information about the biodiversity of the city of Winamac, Pulaski county, Indiana, and the ecological aspects that emerged by the processing of the collected data. The application will make it available to the user an interactive explorative map through which to read positions, height, diameter, address, name of the specie and dieback percentage of the trees in Winamac. Interactive exploratory graphs will be present in the App and Users will be able also to contribute in collecting data. More detailed information about the project can be found in the RASD document.

The Design Document is a document to provide documentation which will be used to aid in software development by providing the details for how the software should be built. Within the Document are narrative documentation of the software design for the project including use case models and other supporting requirement information.

1.1 Purpose

The purpose of the Design Document is to provide a description of the design of a system fully enough to allow for software development to proceed with an understanding of what is to be built and how it is expected to be built. The Design Document provides information necessary to provide description of the details for the software and system to be built.

1.2 Scope

This Design Document is focused on the base level system and critical parts of the system and on the generation and modification of its structure. The system will be used in conjunction with other pre-existing systems and will consist largely of a document interaction facade that abstracts document interactions and handling of the document objects.

1.3 Definitions, Acronyms and Abbreviations

- HTML: **Hypertext Markup Language** (**HTML**) is the standard markup language for documents designed to be displayed in a web browser. It can be assisted by technologies such as Cascading Styles Sheets (CSS) and scripting languages such as JavaScript. Web browser receives HTML documents from a web server or from local storage and render documents into multimedia web pages.
- Py: **Python**, is an interpreted, high-level, general-purpose programming language.
- DB: **database,** an organized collection of data, generally stored and accessed electronically from a computer system. Where databases are more complex they are often developed using formal design and modeling techniques. The database management system (DBMS) is the software that interacts with end users, applications, and the database itself to capture and analyze the data.

- URL: **Uniform Resource Locator (URL)**, colloquially termed a web address, is a reference to a wen resource that specifies its location on a computer network and a mechanism for retrieving it
- SQL: **Structured Query Language (SQL)** is a programming language that is typically used in relational database or data stream management systems.
- HTTP: Stands for "**Hypertext Transfer Protocol.**" HTTP is the protocol used to transfer data over the web. It is part of the Internet protocol suite and defines commands and services used for transmitting webpage data.
- Diameter at breast: **Diameter at breast height**, or **DBH**, is a standard method of expressing the diameter of the trunk or bole of a standing tree. DBH is one of the most common dendrometric measurements. Tree trunks are measured at the height of an adult's breast, which is defined differently in different countries and situations. In the United States, DBH is typically measured at 4.5ft (1.37m) above ground.
- Dieback percent: a condition in which a tree or shrub begins to die from the tip of its leaves or roots backwards, owing to disease or an unfavourable environment. Dieback is measured as the **percent of branch tips in the crown that are dead**; Dieback also can be described as none (0-5%), light (6-20%), moderate (21 -50%), or severe (51 -100%).
- WGS84 crs: **coordinate reference system (CRS)** is a coordinate-based local, regional or global system used to locate geographical entities. **World Geodetic System 1984 (WGS84)** is an Earth-centered, Earth-fixed terrestrial reference system and geodetic datum.

1.4 References

- Documents Utility Software Development Plan
  - Version 1.0, Last Updated on 2020-05-10

1.5 Overview

The Design Document is divided into 6 sections with various subsections. The sections of the Design Document are:

1 Introduction
2 Structure of the database
3 Structure of the Software: database script
4 Structure of the Software: Flask Application
5 Structure of the Software: other functional programs
6 Use cases and requirements mapping on the components of your software
7 Organization of the team

2.DataBase

2.1 Structure of the DataBase

In this first chapter we're going to define and describe the structure of the DataBase we're going to implement in our Web Application.
First of all databases are defined as a set of structured data.
The user can acquire data present in the database by interacting with it through a DataBase Management Systems (DBMSs), which is an application that allows the user to efficiently access the data present in the database, by using specific tools that allows to create a connection with DataBase and send specific requests in order to query the data.
It guarantees a direct access to data based on their properties and it also offers languages to describe data and queries on them in a way that is independent on the way they are physically stored.

In particular, in this Web application, we have chosen to use PostgreSQL as a DataBase Management System. As its name suggests, PostgreSQL is written in SQL language, which is a Declarative Query Language,

- **DDL – Data Definition Language**: useful for defining database structure (relation schemas) and data access control.
- **DML – Data Management Language**: useful for query, read, update and delate tuples (CRUD operations).

The interaction between the Web Application, written in Python, with PostgreSQL is handled by a specific Driver, in particular we have chosen to use as Driver the psycopg2 library. Through this library it's possible to perform all the four steps of interaction:

1) Open the connection whit the DB Server with the command connect().
2) With the command cursor() it is possible to create a cursor, which is the mechanism that allow us to send requests in SQL to the DataBase trough the command execute().
3) Commit the operation with the command commit().
4) Close the connection with the DataBase server through the command close().

In order to initialise the DataBase and create the tables that we're going to use in our Web application we're going to use a specific python code that through the driver psycopg2 is able to:

- Drop all the existing tables in the DataBase that has the same name of the one we're going to use in our Web Application.
- Create two new empty tables, one that contains the information regarding the registered users and one containing the information of the comments that the users can leave and one that contains the geospatial data entered by the users.

2.2 Specific table inside our DataBase

In order to access to the contents of the Web Application is necessary for the user to be registered, so the application must allow the registration of users, through the insertion of specific personal information (username and password) in the DataBase, in order to allow the creation of a public profile.

Therefore, the first table we have defined is the one that contains the information about the users. The user will be asked to enter a username and a password, each new user will be associated with an userID number when registering, userID number is unique for each user and allows the identification the user, creation of session and allows users to visualize specific contents of the Web Application.
The application gets the information sent by the user that wants to register (username and password), it opens a connection with the DataBase, in order to check through a query if the username exists in the database. If it does not exist the system accepts the new registration and saves the user's information in the Database. Once the operation is concluded, the system and redirects the user to the login page.
Every already registered user will have to log in every time he / she goes on the Web Application.
Every time the system has to get the information sent by the user client (username and password) and check if the username exists in the database. If it does the system stores the user id in the session variable, if not the system will send an error message to the user.

USER	
user_id	SERIAL PRIMARY KEY
user_name	VARCHAR(255) UNIQUE NOT NULL
password	VARCHAR(255) NOT NULL

Once the user is logged in, as well as extract custom views of the data, he can also leave comments, suggestions and opinions, in order to share information with other users, but also to help us to develop and improve our web application. It is necessary acquire the information from the client, check that they are complete (if not send an error message) and store the post in the database. The post will have a title and a body.

POST
post_ID SERIAL PRIMARY KEY
author_ID INTEGER NOT NULL
Created TIMESTAMP
Title VARCHAR(350) NOT NULL
Body VARCHAR(500) NOT NULL
FOREIGN KEY (author_ID)
REFERENCES blog_user(user_id)

The system is an informative application to provide users a common platform through which to access data on trees in Wimanac.
The data collection we have chosen from Epicollect provides the users information about the trees: it's geographic position, name of the species, condition, diameter, date of creation of the data, height, dieback percentage. However, the user can also contribute in data collection, by inserting in the system new data that he has personally collect. Those data will have the same attribute of the ones present in Epicollect and will be stored in a table present in the DataBase.

DATA	
Data_ID	SERIAL PRIMARY KEY
Author_ID	INTEGER NOT NULL
Created	TIMESTAMP DEFAULT NOW()
Species	VARCHAR(350) NOT NULL
Dieback	INTEGER NOT NULL
Diameter	INTEGER NOT NULL
Height	INTEGER NOT NULL
Latitude	VARCHAR(350) NOT NULL
lONGITUDE	VARCHAR(350) NOT NULL
FOREIGN KEY	(Author_ID)
REFERENCES	data (data_id)

3. Structure of the Software: Flask Application

After the creation of the database the next step is to build the software that allows to insert data in it, retrieve and work with them, implementing all the use case identified.

The Flask Application consists of the app logic part, the Py code, and the template part. It interacts with the web server through the wsgi, whose purpose is to handle processing requests from the web server and decide on how to transfer those requests to the application framework, and communicates with the database server to ensure that all the information is persistent in it.

The templates control the overall look and layout of a site. They provide the framework that brings together common elements, modules and components as well as providing the cascading style sheet for the site.

The app logic of the software is completely unbound from the template part, in fact the first part has the scope of retrieve and make available the data, instead the second one deals with the representation of the data in the web page. What links the two parts is the template engine, that inserts the information provided by the app logic and fits them instead of the fixed marker.

For each use case is needed to define the Python code and the template, needed for the interaction with the users.

3.1 Template part

The template part includes the templates that support the Flask App and the one that support the program for the visualization of the plots. As had been already said the templates are HTML code that have the purpose to control the presentation, the style and the organization of the elements of the web page.

Each page in the application will have the same basic layout around a different body. Instead of writing the entire HTML structure in each template, each template will *extend* a base template and override specific sections; base template is so the first template. There are three blocks defined here that will be overridden in the other templates:

1. {% block title %} will change the title displayed in the browser's tab and window title.

2. {% block header %} is similar to title but will change the title displayed on the page.

3. {% block content %} is where the content of each page goes, such as the login form or a blog post.

The index template is used to represent data once we create a URL route in our app.

At this preliminary step what is visualized is the title and the menu bar, as specified in the nav section of the code, with the possibility to do the logout, if he is already in session, or to Register or procede with the Login.

The Register and Login functions are supported by similar templates: in the code there are the block header containing the title and the block content that contains the labels and input values, which are the username and the password, required for the two functions.

Once the user is logged in the blog the software uses another index template.

In the block header on find the function to give the possibility to the user to create a new post, instead in the block content is checked that, if the user fits with the user of one of the posts that already exist, he can update his post.

The template for the create post function includes inside of the block content the labels forthe title and the body of the post, with the two inputs to be insert, and the input save to submit the post.

The template used for the update has a similar structure with more the form action to delete the post, with the delete input and the onclick return to give the possibility to the user to confirm the elimination.

The HTML template that support the Flask python code for the integration with Bokeh for the dynamic visualization of the plots includes import of an URL for the style and, inside the body, the text to be visualize with the settings of allignment and colour, the weight and the size of the used font.

3.2 Python code

Purpose of the Python code is to answer to all the requests coming from the user through the web server, with whom it interface through the WSGI, to collect inside all the available functions and recover all the data needed by the database server.

The packages we use in our code are:

- **Flask**: lightweight WSGI web application framework, from it we import:

1) Flask itself
2) Render-template, a template engine that choose what template utilizzare in base alla richiesta
3) Request, with which Flask creates a request object based on the environment it received from the WSGI server
4) Redirect, to redirect a user to another endpoint

5) Abort, to abort a request early with an error code
6) Flash, to flash a message, passed as argument of the function flash(), to the user
7) Url_for, to create dynamic URLs, preventing the overhead of having to change URLs thoughout an application
8) Session, used to set and get session data, it works like a dictionary and the data are stored in the browser as a cookie
9) g, standing for global, is a simple namespace object that has the same lifetime as an application context and use the session or a database to store data across request

- **Werkzeug.security,** used to generate password hash and store them securely; for these purposes we import:

1. check_password_hash
2. generate_password_hash

- **Werkzeug.exceptions**, module that implements a number of Python exceptions the user raise from within his views to trigger a standard non-200 response. We import:

1. abort

- **Psycopg2**, module used to connect to a PostgreSQL database; we only import the function connect

- **Bokeh**, an interactive data visualization library to visualize data and create interactive plots and applications running on a web browser; in particular, from Bokeh.embed we import:

1. server_document, functions for embedding Bokeh standalone and server content in web pages
2. subprocess, to allows the bokeh app running on port 5006 to be accessed by Flask at port 5000

Once all the packages and module are imported, in our app logic are present all the functions defined for each use case and need that the user can request in his session.

First there is the creation of the application instance through the Flask function, to which we pass as argument the name and the template folder that we have implemented for our app; so we set a secret key to some random bytes.

Then we test the connection with the database and define two functions in order to distinguish the two possible cases:

- **get_dbConn:** if we are not connected yet, we pass to the function the txt file of our DB configuration containing dbname, username and password, the function read (readline) the text, save the string and connect with our db

- **close_dbConn**: if connected, the function disconnects from our db, closing the connection and removing the attribute (pop)

All the other functions present in the code fall in cases in which the is the user that can do a request to the server, so for each of these we preliminary declare the @app.route to map the specific URL with the associated function that is intended to perform some task; with @app.route('/') we create a URL route in our application and if we visit the particular URL mapped to some particular function, the output of that function is rendered on the browser's screen; route() decorator tell Flask what URL should trigger our function.

By default a route only answers to GET requests, so in order to the function we are implementing and the case we are in we use the methods argument of the route decorator to handle different HTTP methods (GET, POST, if present in the template)

The functions that we have in our python code are:

REGISTER: once we have created the app route, we define the function (def register). The first case that we consider is that the user want correctly to register (POST method) and he insert the requested inputs of username and password (request.form); after this (if error is None) there is the connection with the DB, and the execution of the insertion (INSERT INTO) of the user's data in the db table WITH A SQL request (password is encrypted). Then we close the connection with the db and redirect the user to the login page (url_for('login')); if all works correctly the return of the function is the redirection to the login template (redirect(url_for('login')).

Error cases that we evaluate (if error is not None) are the possibility that the user doesn't insert one of the required input, in this case a message warns him to insert the missing information, and the case in which the user is already registered; in this case we open the connection with the db and we check if the user_id inserted is present yet in the user's table. In this case a message warns the user that he is already registered and the connection is closed.

For all the other cases of error a flash message of error is sent, and the return of the functions is the register page (render_template)

LOGIN: structure of this function is similar to that of the register case. In this case the user insert his data (request.form), the connection with the database is open and there is a comparison between the inserted username and those present in the db; we close the connection and, if there are no errors (error is none) a session is created for the user (session module) and he is redirected to the index page. Error cases can be that the user had insert an incorrect information, and in this case a message warns him. If the error persists a flash error message is sent and the return is the login template.

LOGOUT: logout function only delete the session already created (session.clear()) and redirect the user to the index page.

LOAD_LOGGED_IN_USER: this function is a callback used to reload the user object from the user ID stored in the session. It takes the unicode ID of a user (session.get('user_id')) and return the corrisponding user object; if there are no user in session (user_idis None) it returns False, either it

connects to the db, it retrieves all the data corresponding to that user; then the connection is closed and the return is True.

INDEX: index function gives the possibility to the user in session to visualize all the data present in the web page. It is activated when the user accesses the app home page and it extracts from the database all available posts and visualize them on the screen.

Connection with the db is open, an SQL request is get to retrieve all the posts and and all the row of the query result are saved (posts = cur. fetchall(); connection is closed and for the user in session (load_logged_in_user function is recall) the return is to the index page that contains all the posts retrieved.

CREATE: this piece of code gives the possibility to the user in session to create and publish a post in the blog. In the definition of the function we specify that if the user is in session (function load_logged_in_user is recall) and he wants to publish a post (request.method == 'POST') there is no error (error is None), title and body of the post inserted by the user (request.form) are acquired (in case of missing input an error message is sent and in case of persistence the user is redirected to the index page), connection with the db is opened so that the post can be saved in the table (INSERT INTO request); after that connection is closed and the user is redirected to the index page.

Errors that can occur are that the method is not POST but GET (in this case the user return to the create page) and that the function load_logged_in_user returns False; in this case a flash error warns the user that insertion of the posts is reserved only to loggedin users, and the return is the redirection to the login page.

GET_POST: this function give the possibility to the user to retrieve post by the title passed in the argument of itself. A connection with the db is performed and with an SQL request we retrieve and save the row, between all those that are present in the post table, that represents the wanted post, closing at the end the connection.

If the post was not found the operation is abort (abort404) and a message warns the user, instead if the post found is different by that the user was looking for the operation is abort in the sameway (abort 403).

UPDATE: update function allows the user in session to update a post of him that he had previously published. The structure of this function is similar to the CREATE ones, so we check if the user is in session and if the method request is the POST one; if there is no error the function execute an UPDATE SQL request and set the post whose title is equal to the id passed as argument of the function.

What differentiates this function from the create one is that, as had been said yet, when we connect to the db the SQL request is not of INSERT but of UPDATE the post choosen; then, after checking if the user is in session or not, we recall the get_post function to retrieve the post that the user wants to update and we save it in a variable.

DELETE: this function allows the user to delete a post published in the blog, whose id is passed as argument of the function; what this piece of code do is to open a connection with the db and execute

an SQL DELETE request, deleting the post from the table whose id correspond to that passed by the user. At the end the connection is closed and the return of the function is to redirect the user to the index page.

BASH_COMMAND: in order to let the user visualize our multi-plot, map and bar, we define this function and use the Bokeh module called subprocess to allows the bokeh app running on port 5006 to be accessed by Flask at port 5000; then we open a route with the @app route ('/') and with the server_document module it embed Bokeh standalone and server content in web pages.

At the end of the code is specified that if it's running in stand alone mode, application can be run.

4. Structure of the Software: other functional programs

The components described in 4.1 and 4.1 paragraphs are Python libraries, they are used during the development of the codes for the creation of the templates. They allow you to process Epicollect5 data in a simpler way, in order to create the html files which will then be used in the Python code. Below, a detailed description of both programs.

4.1 REST API JSON data request

REST is the acronym for REpresentational State Transfer. It is architectural style for distributed hypermedia systems. API is the acronym for Application Programming Interface. It is a set of rules that allow programs to talk to each other. The developer creates the API on the server and allows the client to talk to it. REST determines how the API looks like. It is a set of rules that developers follow when they create their API. One of these rules states that you should be able to get a piece of data (called a resource) when you link to a specific URL. Each URL is called a request while the data sent back to you is called a response.
JSON uses human-readable text to store and transmit data objects consisting of nested attribute–value pairs and array data types (or any other serializable value).
Purpose of the program is to perform a REST API JSON data request with Python; we use the requests library to send the request to the REST API service of the Epicollect5 platform by exploiting a sample public citizen science project.
The packages imported in the program are:

- **requests:** Requests is an Apache2 Licensed HTTP library, written in Python. Requests will allow you to send HTTP/1.1 requests using Python. With it, you can add content like headers, form data, multipart files, and parameters via simple Python libraries. It also allows you to access the response data of Python in the same way.
- **Json**: JSON (JavaScript Object Notation) uses human-readable text to store and transmit data objects consisting of nested attribute–value pairs and array data types (or any other serializable value).
- **pandas:** pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with structured (tabular, multidimensional, potentially

heterogeneous) and time series data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python.

- **geopandas**: 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.

- **shapely:** Python package for manipulation and analysis of 2D geospatial geometries

  First the program retrieves data from the Epicollect5 platform. Using the get module of requests and inserting as parameter the http link of the project it launch the request and has as result the csv data that represent the entries of the project. The text data are saved in a variable and with the use of the loads module of Json it turns JSON encoded data into Python objects. Then with the json_normalize function of pandas the program normalize semi-structured JSON data into a flat table; once the table is ready, using the pd.to_numeric module the program converts all the characteristics of interest,passed as argument, to anumeric type: Latitude, Longitude, Diameter at breast, Dieback percent, Height, Species, Address, Condition.
  Now the program. using the gpd.GeoDataFrame, transform the DataFrame into a GeoDataFrame and plot it. A GeoDataFrame object is a DataFrame that has a column with geometry.
  From shapely.geometry is imported Point to represents geometric objects consist of coordinate tuples, then the GeoData are saved in the variable TREES. With a for cycle for i=0 to the number of entries of interest with the .loc property of pandas, that allows to access a group of rows and columns by labels, it retrieve all the attributes by the DataFrame of each entries, save themin temporary variables and save this values in the TREES GeoDataFrame.
  At the end of the code the program assign the WGS84 crs to the TREES values through the .crs moduleof GeoPandas and the shapefile of the Point is saved in local using the .to_file function of GeoPandas to save GeoDataFrame.

4.2 Map-plot

Purpose of the program is to show in the web page a spatial data plot of the data saved in local in the REST API JSON program with Bokeh.
The packages used in the program are:

- **geopandas as gpd**
- **from bokeh.models it imports ColumnDataSource** (to share data between multiple plots and widgets, such as the DataTable) **LabelSet** (to set annotations)
- **from bokeh.plotting it imports figure** (to create a new figure for plotting), **show** (to show the results), **output_file** (output to static HTMLfile)
- **from bokeh.tile_providers it imports CARTODBPOSITRON** (tile source for CartoDB Tile Service)

The program first retrieves the shapefile using the read_file.to_crs of the GeoPandas package, then define a getPointCoords function that extract coordinates from the geodataframe and calculates

coordinates ('x' or 'y') of a Point geometry; if the type of the coordinate is ‘x’ it returns the x rows, elseif it returns the y rows.
Through the DataFrame.apply of pandas the program apply the function getPointCoords (that returns a table of Path instances) along the axis 1 of the DataFrame; then it saves the coordinates as attributes in a new dataframe, use the dataframe as Bokeh ColumnDataSource and specify feature of the Hoover tool.
Once the variables are defined the program create the Map plot. Using the figure module it creates a new figure for plotting, with settings of range passed as arguments, that is saved in the variables p1; then, with the add_tile function it add a basemap tile. At the end of the code the program add finally Glyphs and Labels and add them to the plot layout: then it output the plot and show(p1).

5. Use cases and requirements mapping on the components of your software

Use case 1: User’s registration

The use case "user’s registration" is represented in the code by a function: **register( )**.

This function works through two HTTP requests to transmit information: **POST** (it interacts with the server but does not save the parameters in the URL, so it is useful for recording with personal data) and **GET** (It is simpler and more immediate; it is used in those requests where it is useful to save the required parameters in the URL, for example to be cachati. A classic use case is a search query).

POST:
- Get the information sent by the user client (username and password).
-Check that both have been entered. If either is missing, report error "username / password is required".
-If both are inserted, connect to the database with the function **get_dbConn (),** give the position on Database to a parameter (curr) and execute the query to search the inserted username.
-Check if the username already exists with the **fetchone()** function. If it already exists give the message "User is already registered".
-Close the connection with the database using the function **close().**
-If it does not exist, connect to the database, run the query to enter the new user and save with the **commit ()** function.
-Use the function **redirect()** to redirect the user to the login page.

GET:
-Use the **render_template()** function has to send the user on the registration page

Use case 2: login

This use case is represented in the code by **login()** function. Also this function works through the HTTP POST request and the HTTP GET request.

POST:

- Get the information sent by the user client (username and password).

- connect to the database with the function **get_dbConn (),** give the position on Database a parameter (curr) and execute the query to search the inserted username.

- Use the function **fetchone()** to check if the username is correct.

-Close the connection and save with **commit()** function.

-If username is not correct send an error message to the user.

-Check if password is correct with the function **check_password_hash()** . If it is not correct send an error message to the user.

-If both are correct : clean the session variable with the function **clean(),** store the user_id in the session variable and the redirect user on the index page.

GET:

-Use the **render_template()** function has to send the user on the login page.

Use case 3: logout

This use case is represented in the code by **logout()** function.

- Clean the session variable with the function **clean().**

-Redirect the user on the index page (main page).

Use case 4: leave a comment

This use case is represented in the code by **create()** function. Also this function works through the HTTP POST request and the HTTP GET request.

-Firstly check if the user is logged in.

POST:

-If the user is logged in (its ID is in the session), acquire information from the client saving them into variable (comment).

- Check that they are complete (if not send an error message).

-Connect the database and execute the query to store the comment in the database.

-Close the connection and save with the **commit()** function.

-Redirect on the main page.

GET:

- if user is logged in, render the creation page
- If user is not logged, the system sends an error message to the user and redirect him/her to the login page.

Use case 5: update a comment

This use case is represented in the code by **update()** function. Also this function works through the HTTP POST request and the HTTP GET request.
-Firstly check if the user is logged in.
-This function receives a parameter: the id of the post to be updated

POST:
-If the user is logged in, check whether the post with the specified id exists.
-if it doesn't exist, send an error message and redirect the user to the main page.
-If the post exists, connect to the database to search and update the comment.
-Close the database and save the operations with **commit()** function.
-Redirect to the main page.

GET:
-if user is logged in, render the update template
-if user is not logged in, send an error message and redirect him/her to the login page

Use case 6: delete a comment

This use case is represented in the code by **delete()** function. This function works only through the HTTP **POST** request.
-Connect to the database and execute a query to delete a comment.
-Redirect on the main page.

Use case 7: Visualization of data on a map

This use case consists of plotting the data (previously saved in a shapefile) and adding a basemap.

Create a shapefile starting from epicollect5:
-Send a GET request using the function **get()** from the library "request" and store the raw text of the response in a variable.
- Encode the raw response text into a JSON variable with the function **loads()** from the package "json".
-Use the function **json_normalize()** to fit the variable to a Pandas Data frame.
- Create new columns with numeric values, one for each feature (latitude, longitude, diameter, height, species, address, condition and dieback).
- Transform the DataFrame into a GeoDataFrame and plot it.

-Set the CRS and save all in a shapefile using the function **to_file().**

Then use to **Bokeh** (It is an interactive data visualization library to visualize data and **create interactive plots and applications running on a web browser**).
-Open the shapefile
- Create a function to extract point coordinates (x,y) from the geodataframe and save them as attributes in a new one dataframe.
-Use the dataframe as Bokeh ColumnDataSource (this function maps names of columns to sequences or arrays).
-Create a map plot in the chosen CRS and add title, labels and glyphs to it.
-Tell Bokeh where to generate output with the function **output_file()** , then use the function **show()** to save the graph in an HTML file and view it in a browser.

Use case 8: calculate and visualize data distribution

This use case takes advantage different libraries.
-Firstly check if the user is logged in.
-Using the **Seaborn** library plot data distribution. User can Plot multiple histograms with custom styles.
-Using **bokeh** library visualize a pie char in which is underlined distribution in terms of species.
-Using **bokeh** library visualize a dynamic bar plot in which is underlined distribution of every attribute's data in every road.

Use case 9: upload data

This use case is represented in the code by **upload()** function. Also this function works through the HTTP POST request and the HTTP GET request.

POST:
-If the user is logged in (its ID is in the session), acquire information from the client saving them into variable (comment).
- Check that they are complete (if not send an error message).
-Check they are written in a correct way.
-Connect the database and execute the query to store the features in the database.
-Close the connection and save with the **commit()** function.
-Redirect on the main page.

GET:
- if user is logged in, render the upload page
- If user is not logged, the system sends an error message to the user and redirect him/her to the login page.

7. Conclusion

The purpose of the design document is to provide high level information on the structure of the software and to direct the development of the web app application.
After having set the guidelines, we have split the work in the following way:
Chiara focused on the structure of the database in terms of the tables we have defined, Marianna described the structure of the software we have used to create the database, Gabriele took care of the description of the Flask Application's structure and Martina has associated the related functions to each use case, making a description of it.

As for the programming part, our group is committed to setting weekly goals. We start from the more general functions up to the more specific ones. We have created the skeleton of the app by developing the functions of registration, login, logout and display of the data obtained from Epicollect5.
Then we divided the work as follows: Gabriele took care of developing a function that allows the user to leave comments. Martina focused on the creation of a function to display statistical data through graphs. Marianna developed a program that allows the user to contribute to the collection of data by inserting new ones while Chiara concentrated on developing a function that creates interactive geographical maps using the available data.

As for the test part, we tested the software's behavior: we checked that the program was built correctly (verification) and that it met the user's needs (validation). We made sure that every page of our application is viewable only by logged in users, otherwise a login request message appears. For each function available in our application we have then defined an oracle, that is, stating which response should have a given request and we checked that it matched the actual response of the software. The application therefore sends consistent responses to user requests. Also, the statistical and geographic information provided is consistent with the data downloaded from Epicollect5.

The group undertakes to respect the dates set for the expiry of the various documents and will try, despite the situation, to work in a coordinated and collaborative way by exchanging ideas and suggestions using github as a sharing platform.