■ ***Code freely available but please consider Availability and Requirements section below.***

PRD: AN INTEGRATED DATABASE SOLUTION FOR MEDICINAL AND AROMATIC PLANT RESEARCH IN PHARMACEUTICALS AND TOXICOLOGY

Onur Kenan ULUTAŞ[a]*, Tuğba GÜNBATAN[b]

[a] Department of Toxicology, Faculty of Pharmacy, Gazi University, Ankara, Türkiye, E-mail: onurkenan@gmail.com, ORCID ID: 0000-0001-8819-9461

[b] Department of Pharmacognosy, Faculty of Pharmacy, Gazi University, Ankara, Türkiye, E-mail: tugbagunbatan86@yahoo.com, ORCID ID: 0000-0002-1138-3145

The project aims to create a data management system for plant information by developing a standardized file and data structure for each laboratory, based on the diversity of the findings. The system is designed to allow the user to easily add new data, preserve data integrity, ensure comprehensibility, and eliminate unnecessary challenges. The system involves creating a folder for each plant, where Excel documents containing data for that specific plant are stored. Within each Excel document, cell A1 contains the plant's Latin name, cell A2 contains the relevant data, cell A3 contains the data date, and cell A4 contains the source information. The Python pandas library is used to load these Excel files for data access, and an interface is created to allow users/clients to search for plant names and view the corresponding data. The system is built using tools such as Python, pandas, and the Django web framework. It is important to note that while the system may eliminate the issue of multiple Excel documents and ensure updatability, it does not address the problem of data security and accessibility, and the documents should be maintained and secured under the control of the responsible personnel on a single computer.

Based on the project requirements described, there would be several steps involved in writing a software program to manage the data as follows:

Step 1. Creating a folder structure for each plant and store the corresponding Excel files in their respective folders.

The user should create a folder on the computer for each plant

The user should name the folder after the plant's Latin name, for example, "Lavandula angustifolia Mill."

The user should store all Excel files containing data for that specific plant inside its folder

To create a folder structure for each plant, the user can use the operating system's file system operations in the chosen programming language. For example, in Python, the user can use the os module to create directories using the os.mkdir() function, such as:

```
import os

# create a folder for the plant Lavandula angustifolia Mill

plant_name = "Lavandula angustifolia Mill."

folder_name = plant_name.replace(" ", "_")

os.mkdir(folder_name)
```

This code will create a folder on the computer with the name "Lavandula_angustifolia_Mill." in the current working directory. The user can modify it to create folders in a specific location if needed.

The user should write this code in any Python IDE or text editor, such as PyCharm, VSCode, or Sublime Text. Once the user have written the code, the user can save it with a .py extension and run it using the command prompt or terminal.

Step 2. Labeling each Excel file with the plant's Latin name, data name, data date, and source information.

For Step 2, the user should write a Python script that uses the os library to create a folder for each plant. The name of each folder should be the Latin name of the plant. A continuation of the code example above:

```
import os

# Create a folder for Lavandula angustifolia Mill.

folder_name = "Lavandula angustifolia Mill."

os.mkdir(folder_name)
```

The user can use this code snippet in a loop to create folders for all the plants in the database. The loop should iterate over the list of plant Latin names, and for each name, create a folder with the corresponding name. Here is an example code snippet for creating folders for multiple plants:

```
import os

# List of plant Latin names

plant_names = ["Lavandula angustifolia Mill.", "Salvia officinalis L.", "Rosmarinus officinalis L."]

# Create folders for each plant

for name in plant_names:

os.mkdir(name)
```

This code will create three folders named "Lavandula angustifolia Mill.", "Salvia officinalis L.", and "Rosmarinus officinalis L." in the current directory.

The user can write this code in a Python script file using any text editor or integrated development environment (IDE) of the choice, such as PyCharm, VS Code, or Sublime Text. Make sure that the user have already installed the necessary libraries such as Pandas and have access to the Excel files containing the data for each plant. Once the user have written the code, the user can save the file with a meaningful name and run it from the terminal or within the IDE.

Both step 1 and step 2 involve creating a folder for each plant and storing the Excel files containing data for that specific plant in those folders.

The difference between the two steps is that step 1 involves manually creating the folders and placing the Excel files into those folders, while step 2 involves using Python code to automate this process.

Step 2 uses the os module in Python to create folders and move files programmatically. This can save time and reduce the likelihood of human error compared to manually creating the folders and moving the files.

Step 2 can be used for automation only if the folder structure and Excel file naming convention have already been established manually. If the folder structure and file naming convention

have not been established, then Step 1 would be necessary to create the necessary folder structure and name the Excel files appropriately. However, once the folder structure and file naming convention have been established, Step 2 can be used for automation to load the data from the Excel files into a database.

There is no need to download or install any additional program for creating folders and storing Excel files. This can be done using the computer's file explorer or command-line interface.

To create a new folder, the user can navigate to the desired location using the file explorer, right-clicking on an empty space, and selecting "New Folder" from the context menu. Then, the user can rename the folder to the plant's name.

To store the Excel files in the appropriate folder, the user can simply drag and drop them from their current location to the newly created folder. Alternatively, the user can use the "cut" and "paste" commands to move the files to the appropriate folder.

The Python code for step 1 and step 2 can be written in a text editor or an integrated development environment (IDE) such as PyCharm or Visual Studio Code. Once the user have written the code, it can be saved as a Python file with a ".py" extension.

Step 3. Using the pandas library in Python to load the Excel files and extract the necessary data.

The user will need to create an interface that allows client users to search for plant names and view the corresponding data. This can be done by creating a web application that runs on a server. The web application will need to be programmed using a web framework such as Django or Flask.

Here are the steps that user should follow to create the interface:

Installing a web framework such as Django or Flask.

Python can be found from the official website (https://www.python.org/downloads/)

Installing pip: pip is a package installer for Python. It allows you to easily install and manage Python packages. The user can install pip by running the following command in the terminal or command prompt:

python -m ensurepip --default-pip

Installing the web framework: Once the pip have installed, the user can install the web framework of its choice using pip. For example, to install Django, the user can run the following command:

pip install Django

Verifying the installation: After the installation is complete, the user can verify that the web framework is installed correctly by running a simple test command. For example, to test the installation of Django, the user can run the following command:

django-admin –version

If Django is installed correctly, the user should see the version number displayed in the terminal or command prompt.

With the web framework installed, the user can start building the application.

Create a new web application project.

To create a new web application project in Django, the user can follow these steps:

Open a command prompt or terminal window on the computer.

Navigate to the directory where user want to create the new Django project.

Run the following command to create a new Django project:

django-admin startproject projectname

Replace "projectname" with the name of the project.

After running the command, user will see a new directory with the name of the project that has been created. This directory will contain the basic structure of the Django project.

Navigating to the new project directory using the following command:

cd projectname

Next, the user will need to create a new Django app within the project. To do this, the user should run the following command:

python manage.py startapp appname

Replace "appname" with the name of the app.

After running the command, the user will see a new directory with the name of the app has been created inside their project directory. This directory will contain the files for the app.

Now that the user have created the project and app, the user can start building their web application using Django. The user can define their models, views, templates, and URLs within their app directory and configure their project settings in the project directory's settings.py file.

Finally, the user can start the development server using the following command:

python manage.py runserver

This will start the server at http://127.0.0.1:8000/ where the user can view their web application in a web browser.

This would have successfully created a new Django web application project and can start building the application.

Defining the views for the web application. These views will contain the code that retrieves the data from the Excel files and displays it to the other users/clients.

Opening the views.py file: This file is located in the directory of the Django app, under the app folder. It contains the functions that define the views of the web application.

Defining a view function: To define a view, the user need to create a Python function that takes a web request as its argument and returns a web response. For example, the user can define a function called plant_data that retrieves the data from the Excel files and displays it to the other users/clients. Here's an example code snippet:

```
from django.shortcuts import render

from .models import Plant

def plant_data(request):

    plant = Plant.objects.all()

    context = {'plant': plant}

    return render(request, 'plant_data.html', context)
```

In this code snippet, we are using the Plant model to retrieve data from the database, and then passing it to the plant_data.html template using the context dictionary.

Mapping the view to a URL: After defining the view function, the user need to map it to a URL so that it can be accessed by the other cleints. This is done in the urls.py file, which is located in the app folder. The user can define a URL pattern that maps to the view function the user just created. Here's an example code snippet:

```
from django.urls import path

from . import views

urlpatterns = [

    path('plant_data/', views.plant_data, name='plant_data'),

]
```

In this code snippet, the user is defining a URL pattern that maps to the plant_data view function. When the other clients visit the URL /plant_data/, Django will call the plant_data function to handle the request.

Creating a template: A template is an HTML file that defines how the data retrieved by the view should be displayed to the other clients. The user can create a new template called plant_data.html in the templates directory of the app, and add the necessary HTML and Django template tags to display the data. Here's an example code snippet:

```
{% extends 'base.html' %}

{% block content %}

    <h1>Plant Data</h1>

    <table>

        <tr>

            <th>Name</th>

            <th>Location</th>

            <th>Data</th>

        </tr>

        {% for p in plant %}

            <tr>

                <td>{{ p.name }}</td>

                <td>{{ p.location }}</td>

                <td>{{ p.data }}</td>

            </tr>

        {% endfor %}

    </table>

{% endblock %}
```

In this code snippet, the user is extending a base template called base.html, and using Django template tags to display the data in a table.

These steps have defined a view in the Django web application that retrieves data from Excel files and displays it to the clients using a template.

Define the URLs for the web application. These URLs will be used by the user to access the different views of the web application.

After defining the views for web application in Django, the next step is to define the URLs that will allow clientss to access those views.

The user can define URLs for their Django web application in a file called urls.py, which is typically located in the same directory as the views.py file. In this file, the user will define a URL pattern that will map to a specific view function.

Here are the steps to define URLs for the Django web application:

Open the urls.py file in the text editor.

Importing the views module that contains the functions that the user want to map to URLs. This is typically done by adding a line like this at the top of the file:

from . import views

Defining a URL pattern by creating a new urlpatterns list. This list should contain one or more url() functions that map a URL pattern to a view function. For example, the following code maps the URL pattern "/plants/" to a view function called plants_view:

```
urlpatterns = [
    url(r'^plants/$', views.plants_view, name='plants'),
]
```

In this code, the url() function takes three arguments: the URL pattern (in this case, "/plants/"), the view function (views.plants_view), and an optional name for the URL pattern ("plants").

The user should save the urls.py file and restart their Django development server. After that user should now be able to access the view function at the URL pattern that they have defined. In this example, the user could access the plants_view function by visiting http://localhost:8000/plants/ in their web browser.

Creating HTML templates that define the layout and design of their web application.

After defining the URLs for the web application, the next step is to create HTML templates that define the layout and design of the web application. This should take these steps:

Creating a new folder in the Django project directory called templates. This is where all the HTML templates will be stored.

Inside the templates folder, creating a sub-folder with the same name as the app. For example, if the app is called "PlantData", the user should create a folder called "PlantData" inside the templates folder.

Inside the app folder, creating a new HTML file for each view, the user have defined earlier. For example, if the user have defined a view called plant_list, then the user should create a new file called plant_list.html.

In each HTML file, defining the layout and design of the page using HTML, CSS, and JavaScript. The user can use a variety of tools and libraries to create the HTML templates, such as Bootstrap or Materialize.

Using Django's template language to insert dynamic data into the HTML templates. For example, if the user want to display the data for a specific plant, then the user can use the following code in their HTML template:

```
<h1>{{ plant.name }}</h1>

<p>{{ plant.description }}</p>
```

This will display the name and description of the plant, which will be retrieved from the database using the views the user defined earlier.

Once the user have created the HTML templates, it is need to update the views to render these templates when other clients request a specific URL. This can be done using Django's render() function, which takes the request, the template name, and any data that should be included in the template.

For example, if it is want to render the plant_list.html template for the /plants/ URL, it following code can update the plant_list view:

```
from django.shortcuts import render

from .models import Plant

def plant_list(request):

   plants = Plant.objects.all()

   return render(request, 'PlantData/plant_list.html', {'plants': plants})
```

This will retrieve all the plants from the database and pass them to the plant_list.html template, which will display them on the page.

With these steps, the user can create HTML templates that define the layout and design of the web application and render them using Django's views

Writing the JavaScript code that handles other clients interactions, such as search queries and data filtering.

Creating a new JavaScript file: In the project directory, creating a new file called script.js or any other name of the user choice. This file will contain all the JavaScript code that handles other clients interactions.

Linking the JavaScript file to HTML templates: In each HTML template that requires JavaScript functionality, it is needed to add a link to the script.js file using the script tag. For example:

```
<script src="{% static 'js/script.js' %}"></script>
```

This tells the web browser where to find the script.js file and to load it along with the HTML template.

Using JavaScript to handle other clients interactions: In the script.js file, it is needed to write a JavaScript code that handles clients interactions such as search queries and data filtering. Here are some examples:

Search queries: To handle search queries, the user can use the querySelector method to get the search input element from the HTML template, and then attach an event listener to it. When other clients submit a search query, the event listener can retrieve the query and use it to filter the data. For example:

```
const searchInput = document.querySelector('#search');

searchInput.addEventListener('submit', (event) => {

  event.preventDefault();

  const query = searchInput.value.toLowerCase();
```

```
  // filter the data based on the query
});
```

Data filtering: To filter data based on other clients input, the user can use JavaScript to manipulate the HTML elements that display the data. For example, the user can add a class to certain elements to hide or show the other clients based on their input. Here's an example of how the user can use JavaScript to filter a list of items:

```
const filterInput = document.querySelector('#filter');
filterInput.addEventListener('input', () => {
  const query = filterInput.value.toLowerCase();
  const items = document.querySelectorAll('.item');
  items.forEach((item) => {
    const itemName = item.querySelector('.name').textContent.toLowerCase();
    if (itemName.includes(query)) {
      item.classList.remove('hidden');
    } else {
      item.classList.add('hidden');
    }
  });
});
```

In this example, the other clients can type a search query in an input field with the id filter, and the JavaScript code will filter a list of items with the class item based on whether their name contains the search query. The hidden class is added to items that do not match the query, and removed from items that do match, which hides or shows them respectively.

These steps are examples to write JavaScript code that handles other clients interactions such as search queries and data filtering in the web application.

Once the user have completed these steps, there will be a functional web application that allows other clients to search for plant names and view the corresponding data.

Step 4. Saving the extracted data into a database for easy access and manipulation.

Choosing a database management system (DBMS): The user need to choose a DBMS that supports the data model that they are using. Here it is chosen the MongoDB.

Installing MongoDB: MongoDB, the database management system can be installed from the official website (https://www.mongodb.com/).

Installing PyMongo: PyMongo is a Python library that allows the users to connect to a MongoDB database from Python. The user can install PyMongo using pip, the package installer for Python the user should open the terminal or command prompt and enter the following command:

```
pip install pymongo
```

Connecting to the MongoDB database: To connect to a MongoDB database from Python, the user need to create a MongoClient object and pass the connection string as a parameter. Here is an example:

```
from pymongo import MongoClient

# Establishing a connection with MongoDB

client = MongoClient('mongodb://localhost:27017/')

# Creating a database named 'plant_db'

db = client['plant_db']
```

Defining a collection to store the data: In MongoDB, data is stored in collections, which are similar to tables in relational databases. The user can use the following code to create a collection named 'plants' to store the extracted data.

```
# Creating a collection named 'plants' in 'plant_db'

plants_collection = db['plants']
```

Saving the extracted data to the MongoDB database: Finally, the user can save the extracted data to the 'plants' collection using the insert_many() method provided by pymongo.

```
# Saving the extracted data to the 'plants' collection

plants_collection.insert_many(data)
```

Here, data is the list of dictionaries containing the extracted data from the Excel files.

After this the project will have now saved the extracted data to the MongoDB database for easy access and manipulation.

Step 5. Creating a user interface to allow clients to search for plant names and view the corresponding data.

At this step plant data can be retrieved from the database, but there is need to create a web application to display the data and allow clients to search for plant names. Flask would be a lightweight web framework that is well-suited for this purpose.

To install Flask, the user can open a command prompt or terminal and type:

```
pip install Flask
```

After that, the user can create a new Flask application using the following code:

```
from flask import Flask

# Create a new Flask application

app = Flask(__name__)
```

Creating a route to display the plant data: Next, the user need to create a route in the Flask application that displays the plant data. The user can use a template engine like Jinja2 to render HTML templates with the data.

Here's an example code snippet that creates a route that displays all the plant data in a table:

```
from flask import render_template

# Route to display plant data

@app.route('/plants')
```

```
def plant_data():
    # Retrieve all plant data from MongoDB
    plants = collection.find()
     # Render HTML template with plant data
    return render_template('plants.html', plants=plants)
```

Creating a search form: To allow clients to search for plant names, the user need to create a search form in the HTML template. The form should submit a GET request to a search route in the Flask application.

Here's an example HTML code for the search form:

```
<form method="GET" action="{{ url_for('search_plants') }}">
    <label for="search">Search for plant name:</label>
    <input type="text" id="search" name="query">
    <button type="submit">Search</button>
</form>
```

After setting up a Flask application and connected it to the MongoDB database, the user can create a route that handles search requests using the following code:

```
from flask import Flask, render_template, request
from pymongo import MongoClient
# Set up Flask app and database connection
app = Flask(__name__)
client = MongoClient()
db = client.plants_database
collection = db.plants_collection
# Define route for handling search requests
@app.route('/search')
def search():
    # Retrieve search query from GET request
    query = request.args.get('q')
    # Retrieve plant data from database and filter by search query
    plants = collection.find({'name': {'$regex': query, '$options': 'i'}})
    # Render template with filtered plant data
    return render_template('search_results.html', plants=plants)
```

This code sets up a Flask application and connects it to a MongoDB database. The search function defines a route at /search that handles GET requests with a search query parameter q. The function retrieves the search query from the GET request using request.args.get('q').

The function then retrieves plant data from the MongoDB database using collection.find() and filters the data by the search query using a case-insensitive regular expression match with $regex and $options. The filtered plant data is then passed to a Flask template called search_results.html using render_template().

In order to display the search form to the client, the user will also need to create another route that renders an HTML template with a search form. Here is an example of such a route:

```
@app.route('/')
def index():
    # Render template with search form
    return render_template('index.html')
```

This code defines a route at / that renders an HTML template called index.html, which contains a search form for the client to input their query.

Step 6. Using tools such as Python, pandas, and the Django web framework to build the web interface.

Last step is all about web interface. But first, pandas, which is a powerful Python library for data manipulation and analysis is needed. For this project, pandas can be used to load and manipulate the data retrieved from the database, such as filtering, sorting, and grouping, to prepare it for display on the web interface. It can also be used to perform various statistical analysis and data visualizations. The command to install pandas:

```
pip install pandas
```

Next, to create a Django app for the project.:

```
django-admin startapp myapp
```

This will create a new directory called myapp with the necessary files and folders for a Django app.

At this point defining a view function in views.py that will handle requests to the web interface then in the same views.py file, defining another view function that will handle search requests from clients will be necessary. This function will retrieve the search query from the GET request and use it to filter the plant data retrieved from the database and creating HTML templates for the web interface is also needed.

Obviously, the exact implementation details may vary depending on the specific requirements and constraints of the each laboratory or work group. PRD is freely available and there are no restrictions to use but please consider Availability and Requirements section.

Availability and requirements

• Project name: PRD – Plant Research Database

• Any restrictions to use by academics and non-academics: None but patent pending.

The code for PRD is freely available. It can be used, modified and distributed freely as long as this publication and the original authors are acknowledged. If research projects benefited much from PRD, this publication should be cited in arising papers.