# CowInfo System Structure

By

Li Ju

Gustaf Andersson

Linus Kanestad

# General Structure

The system is designed as a web-application and divided as such with a frontend and a backend. For better portability, the system is built inside Docker containers and orchestrated by Docker Compose.

The frontend utilizes the web application framework known as Django, which is written in Python. This is a very convenient choice since one can easily use other Python packages. In short, one can use Python for preprocessing of data that is then passed and rendered by Django in HTML templates.

The database is constructed based on MySQL. Main functionalities of the back end include preprocessing of files, data insertion, data querying, and metadata management.

The system is running on a Linux platform (RISE internal servers). It is located at **/home/u93007/**.
The following packages are required for the system to work properly:

- **mysql-connector-python**
- **pandas**
- **numpy**
- **django**
- **openpyxl**
- **psutil**

The required packages will be installed inside containers automatically, if Docker and Docker Compose are installed correctly. You can refer here to get the system started.

The directory structure can be viewed in **Directory_Tree.pdf**.

# Frontend

## Navigation Menu

The web application uses a very simple free-to-use side menu bootstrap that is used as means of navigation throughout the site. All of the CSS and javascripts responsible for creating the navigation menu are called in the **base.html** template file, which is later inherited by all other HTML templates used in the web application using a built-in Django function of template-language[1].

## How does Django render a page?

The Django development server (which we are using in production as well) listens after specific URLs to render the page accordingly.
To render a page, it requires the following:

- a *view*-function in **views.py** which is located in the **cowsite** folder. The functions in **views.py** are responsible for rendering the page requested with a specified URL, loading the specified HTML template. The view function returns a HTTP-request, an HTML template, and a Python dictionary variable often called **context.** This dictionary can contain information that is passed to the HTML template, enabling you to render any pre-processed data directly to the template (e.g user feedback messages).
- a URL path in **urls.py** (in in the **cowsite** folder) declared specifically to the view function in charge of rendering that page.
- an HTML-template that is to be called by the view function. The templates for the system are located in **/cow-app/cowsite/templates**.

In short, the Django server listens after specific URLs specified in **urls.py**. This in turn calls the view-function in **views.py** that is connected to that URL. Lastly the view-function returns an HTTP-request to the server which renders the given template together with a **context**.

## POST-requests

The Django server listens for HTTP-requests. These requests can contain data that may come from query choices that the user has put in. These choices are passed back into the system by listening to these POST-requests, which are often initialized by a button press. In our view-function and even the helper functions (see below) we can use the requests caught by the Django server and look **if** it contains a POST-request and more data attached to it. This is how we decide what to render on the page and often what functions to call, by using simple **if** statements to catch POST-request.

[1] The Django template language | Django documentation | Django (djangoproject.com)

## Help-Functions

To render for the proper situations in the view-function, a lot of help-functions have been written that are instead called within the **views.py** file. These help-functions can be found in **/cow-app/functions.py**.

# Backend

The backend is located at **cow-app/src**. It consists of two parts (folders), **apis** and **lib**. Scripts in **lib** are for internal usage within the backend, and they are responsible for database management (**dbmanager**), log management for inserted files (**log manager**), preprocessors for files (**reader**) and insertors for different types of files (**insertor**). Functions in scripts of **apis** are exposed for the front end, so that the front end can call them directly.

## APIs

There are 5 scripts in API part, including **overview.py** for checking the metadata of the system, **query_nl.py** and **query_se.py** for making queries for dutch and swedish data respectively, **scan_nl.py** and **scan_se.py** for scanning directories for dutch and swedish files correspondingly.

Functions in **scan_nl.py** and **scan_se.py** will be called each time one uploads new files via web browser. However, uploading too many large files takes time. An alternative is to manually copy files to the directory named **upload_files/nl** or **upload_files/se** correspondingly, and then upload a random file from the browser to call the **scan** function to insert them all.

In **query_nl.py** and **query_se.py**, some common-used query functions are implemented and exposed to the website already. If complex SQL queries are needed, you can refer to our example codes in R/Python, to connect to the MySQL server and execute your own statements.

## Lib

Scripts and functions in them are for internal usage. All parts are loosely coupled in a moduled fashion. One can modify and extend each component without affecting others.

Let's take insertion as an example to show how the backend works. This will also help you understand the connection between each module better. The workflow of the insertion is:

1. Each time some files are uploaded via the web browser, **scan** function is called from the frontend. Names of all files in the directory will be loaded first.
2. Then the **log manager** will load the log file, in which all inserted files will be recorded. Only files that do not exist in the log file will be inserted, while others will be ignored.
3. Files are categorized by the prefix of their names (that means, file names matter! Be careful renaming files). Non-position files will be inserted immediately while position files will be inserted in the background in new threads.
4. Before insertion, **dbmanager** is used to connect to the database, check if all tables are created and the connection is working. An object for manipulating database is returned.
5. For each file, an **insertor** is initialized according to its file type (position, health, milk info, .etc. ). Together with the database object obtained by step 4, the insertion starts.
6. After the insertion for each file is done, the name of the file will be recorded in the log file by the **log manager**.

# Documentation and other sources

Our report for the course Project in Computational Science (UU-12022) provides other useful information for the system as well. It is attached in <name of our report>.pdf.

## Django

**Official Django Documentation:**
Django documentation | Django documentation | Django (djangoproject.com)

- **Add view-function** (i.e function responsible for loading a page)
  Writing views | Django documentation | Django (djangoproject.com)

- **Pass information to the system using requests** (POST, GET etc)
  Request and response objects | Django documentation | Django (djangoproject.com)
  https://www.youtube.com/watch?v=wzZiONbtwiA

- **HTML Templates** - extend templates etc.
  The Django template language | Django documentation | Django (djangoproject.com)