DOWN TO THE GROUND

Database Documentation

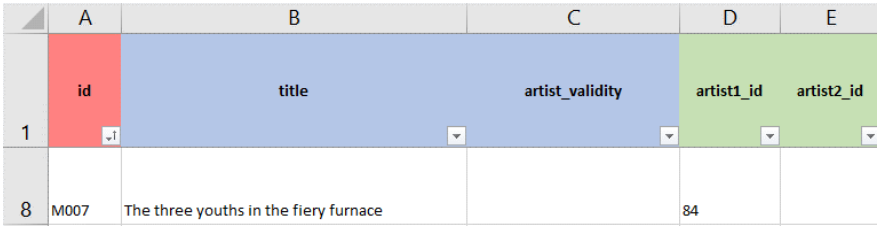
Contents

Backend I: Excel sheet

To keep the method for inputting new data as simple and familiar as possible, it was decided to keep using an excel-file as the functional backend. The original one-sheet excel file was split up into 6 separate sheets, that mimic the eventual SQLite database behind the website:

1. Data
2. Artworks
3. Artists
4. Museums
5. Cities
6. Colours

Splitting up the data like this makes the sheet more legible. The colour of the columns dictates the (possible) relationship of a given field to a field in another sheet (Fig.1):



	A	B	C	D	E
1	id	title	artist_validity	artist1_id	artist2_id
8	M007	The three youths in the fiery furnace		84	

Fig. 1: Excerpt of the [*2. Artworks*] input sheet of the excel

Primary key (PK)	A primary key (PK) is used as a unique identifier to quickly parse data within the table. A table cannot have more than one PK. A primary key's main features are: • It must contain a unique value for each row of data. • It cannot contain null values. • Every row must have a PK value.
Foreign key (FK)	A foreign key (FK) is a field that refers to the PK in another table. By using an FK, different tables can be linked to one another. In this example. The *Artworks* sheet is linked to the *Artists* sheet through columns D and E, that link to the PK (in this case Artist id) in the *Artists* sheet. Instead of having to fill in the name 'Rembrandt' >200 times we simply input the id referencing him. The *Artists* sheet then contains the rest of the relevant information (birthdate, place of birth etc), which limits cluttering of the *Artworks* sheet.
Normal field	These are simply normal fields unique to this sheet, and contain information on the relevant entry.

Data input

To input a new data-entry, we recommend working your way from back to front (starting at sheet 6, ending at sheet 1). This is to ensure all the FK relationships are set up properly.

6. *Colours*

Column name	*Content*	*Styling*	*Required?*
id	PK. Abbreviation of the colour-name	Capitalized., E.g., 'DBl' for 'Dark black'	**YES**
colour_name		First term capitalized. E.g., 'Dark black'	**YES**
hex_code	Used for making renders on the front-end.	Standard hex-code formatting. E.g., #000000	**YES**
group	Allows for grouping of data by colour group.	Capitalized. E.g., 'Black'	**YES**

Most likely, this sheet of the database will remain untouched. It lists the colour-system devised within the DttG project, with relevant hex-codes.

5. *Cities*

Column name	*Content*	*Styling*	*Required?*
city	PK. City name.	Capitalized. E.g., 'Rome'	**YES**
country	Allows for grouping of data by country.	Capitalized. E.g., 'Italy'	NO
continent	Allows for grouping of data by continent.	Capitalized. E.g., 'Europe'	NO
latitude **longitude**	Allows for making geographical data plots. e.g., use https://www.gps-coordinates.net/	Standard lat/lon formatting with 7 decimals. E.g., 55.6867243	NO

This table needs to be filled in properly to ensure FK of other sheets can link to it. There's a high chance that your desired city is already added to the database. Fields that link to this sheet are the following:

- Museum.city
- Artists.place_of_birth
- Artists.place_of_death
- Artworks.place_of_execution

E.g., you will only be able to enter your artwork's place of execution if the city is already filled in in the *Cities* sheet.

4. Museums

Column name	*Content*	*Styling*	*Required?*
id	PK	Integer, automatically increments with 1.	**YES**
museum_name		Capitalized. E.g., 'Chatsworth House Trust'	**YES**
city	FK (to Cities.city)	Same as linked FK relationship. Capitalized. E.g., 'Rome'	NO
website	Museum home-page	https://example-website.org/	NO

3. *Museums*

Column name	*Content*	*Styling*	*Required?*
id	PK	Integer, automatically increments with 1.	**YES**
full_name		LAST NAME, First name. E.g., 'ANTHONISZ, Aert'	**YES**
other_names		LAST NAME, First name | LAST NAME, First name | etc. (From RKD)	NO
Place_of_birth	FK (to Cities.city)	Same as linked FK relationship. Capitalized. E.g., 'Rome'	**YES** (pick unknown if unclear)
Place_of_death	FK (to Cities.city)	Same as linked FK relationship. Capitalized. E.g., 'Rome'	
Year_of_birth		Year ####. E.g., '1612' (From RKD)	NO
Year_of_death		Year ####. E.g., '1612' (From RKD)	NO
Centres_of_activity		City 1 | City 2 | City 3 | etc. (Take from RKD)	NO
Rkd_link		https://rkd.nl/explore/artists/#####	NO
image	???	???	NO

2. *Artworks*

Column name	*Content*	*Styling*	*Required?*
id	PK	M###. 'M-Number' id.	**YES**
title		Title Case. E.g.,'Portrait of a Man'	**YES**
artist_validity		Pick from dropdown menu. Add to 'DROPDOWNS' Sheet if you need other options.	NO
artist1_id	FK (to Artists.id)	Same as linked FK relationship. Integer.	**YES**
artist2_id	FK (to Artists.id)	Same as linked FK relationship. Integer.	NO
place_of_execution	FK (to Cities.city)	Same as linked FK relationship. Capitalized. E.g., 'Rome'	**YES** (pick 49. Unknown if uncertain)
date_validity		Pick from dropdown menu. Add to 'DROPDOWNS' Sheet if you need other options.	NO
date1		Year ####. E.g., '1612'	NO
date2		Year ####. E.g., '1612'	NO
support		Pick from dropdown menu. Add to 'DROPDOWNS' Sheet if you need other options.	NO
medium		Pick from dropdown menu. Add to 'DROPDOWNS' Sheet if you need other options.	NO
height	Height in cm.	Max. 2 decimals	NO
width	Width in cm.	Max. 2 decimals	NO
accession_number			NO
museum	FK (to Museums.id)	Same as linked FK relationship. Integer.	**YES** (pick 49. Unknown if uncertain)
museum_link		URL (preferably permalink)	NO
rkd_link		URL (preferably permalink)	NO
image		???	NO

1. *Data*

Column name	*Content*	*Styling*	*Required?*
id	PK & FK (to Artworks.id)	M###. 'M-Number' id. Links to the relevant artwork entry.	**YES**
no_of_grounds		Integer (1 - ∞)	NO
decsription		Free text	NO
colour_code		Capitalized, from support → paint layer. E.g., 'LY-W'	NO
1	Don't fill in. Automatically assumes the colour of layer1_colour.		NO
2	Don't fill in. Automatically assumes the colour of layer2_colour.		NO
3	Don't fill in. Automatically assumes the colour of layer3_colour.		NO
layer1_colour	FK. (to Colours.id)	Same as linked FK relationship. Colour-code. E.g., 'LY'	NO
layer1_composition		Free text	NO
layer2_colour	FK. (to Colours.id)	Same as linked FK relationship. Colour-code. E.g., 'LY'	NO
layer2_composition		Free text	NO
layer3_colour	FK. (to Colours.id)	Same as linked FK relationship. Colour-code. E.g., 'LY'	NO
layer3_composition		Free text	NO
reliability		Pick between 1-5	NO
sample		Pick from dropdown menu. Add to 'DROPDOWNS' Sheet if you need other options. (Choices: YES/NO/UNCLEAR)	NO
microscopy			NO
elem_analysis			NO
sample_location		Free text	NO
sample_name_1		Free text	NO
sample_link_1		URL (if existing)	NO
sample_name_2		Free text	NO
sample_link_2		URL	NO
researchers		Free text	NO
source		Free text	NO
dttg_new_research		Pick from dropdown menu. Add to 'DROPDOWNS' Sheet if you need other options. (Choices: YES/NO/UNCLEAR)	NO
notes		Free text	NO

Backend II: Django [SQL & Python]

Introduction

After some market-research/literature review and discussions with computer science/database professionals – and taking in mind the researcher's experience with Python – it was decided to use the free and open source back-end web framework **Django.** It is focused on easing the creation of the backend of complex, database-driven websites using python. It is mostly focused on a 'don't repeat yourself' principle, encouraging the coder to use less code, instead creating templates through which data can repeatedly be presented in a similar fashion. Python is used all throughout, for settings, files, and data models.

In practice, this means that we've developed a backend and frontend (full stack) of a website that uses the database to generate both webpages and functions the user can browse through.

Set-up

This is just a basic short explanation of the developed system that presents the user with the data in an interactive fashion. For a deeper understanding, it would probably be necessary to delve into python/html/css and Django-specific terminology through other online resources. More questions? Reach out to: p.j.c.vanlaar@gmail.com

Data-structure

Django runs on an SQL database. SQL stands for Structured Query Language, and is used for relational databases. In essence that means: a collection of tables with defined relationships connecting them (foreign keys).

In the file tree of the Django project, this database is kept in a file called 'db.sqlite3'. However, when first creating a Django project, this database will be completely empty. To be able to fill it, we first needed to define the different tables in the database. Within Django, these are referred to as the different 'Models'. Within the project tree they are defined in the following file: `'dttg_new/data/models.py'`.

In it, the 6 tables (as presented in the Excel) are defined as 6 separate Django models. This ensures we can define properly the type of accepted input (e.g. text, numeric, image) as well as the relationship from a given field to another table.

Data input / Updating the database

To ensure researchers can just keep using the easy-to-use excel sheets for data-input. I've written a python script that automatically runs through the excel sheet and inputs the data into the db.sqlite3 file. Following the methodology/standard also used in the IPERION-CH project – it was decided to delete the entire database with every update. In practice this will mean that 1 main excel sheet will remain the 'true' updated version. This developed Django project is simply a presentation of that data, and should not be considered the true back-up of it.

To update the database, export each of the 6 sheets of the excel as a csv and put them in the `'dttg-new/data/csv'` folder with the names `'colours.csv'`, `'cities.csv'`, `'museums.csv'`, `'artists.csv'`, `'artworks.csv'`, `'data.csv'`. Then, using a python shell run the relevant `.py` scripts in the `'dttg-new/scripts'` folder. This will delete the current stored data in that model, and update it with the data provided in the `.csv` file.

Like entering a new field in the excel, this should happen from back to forth (colours-cities-museums-artists-artworks-data), to ensure the foreign key relationships are set up correctly. E.g., you can't enter a data-entry before the relevant artwork is entered etc.

It is not necessary to delete all data to update a single model. For example, the colours/cities/museums models do not (necessarily) have to be updated when a new artwork is added.

URLs & Views

Now that the data is put into the SQL back-end of the Django-project, it is necessary for a user to be able to interact with it. This happens (mostly) through the `'urls.py'` and `'views.py'` files.

`urls.py`

If a user enters our website and goes to *https://domain.com/* , it sends a request to the `'dttg_new/dttg_new/urls.py'` file. Under the heading `urlpatterns` it will find the following function: `path('', include(data.urls'))`. The first part of this expression (`''`) refers to the ending of the user-requested URL. The second part, then, dictates what the back-end should do with this request. In this example, it refers the user to another file: `'dttg_new/data/urls.py'`.

```
urlpatterns = [
    path('', include('data.urls')),
    path('admin/', admin.site.urls),
]
```

Fig. 2: `dttg_new/dttg_new/urls.py`. Lists the admin page (`domain.com/admin`), as well as links to

In a similar way, if the user would go to *https://domain.com/admin* , it would find `path('admin/', admin.site.urls)` in the `urls.py` file, redirecting the user to the admin log-in page.

In the `'dttg_new/data/urls.py'` file, you can find all the accessible URLs of our webpage. If a user tries to access a URL that is not listed in this file, they would receive a '404-page not found' error.

```
urlpatterns = [
    path('', views.home, name='dttg-home'),
    path('about/', views.about, name='dttg-about'),
    path('artists/', ArtistListView.as_view(), name='artists-overview'), #NEW LIST VIEW
    path('artists/<int:pk>/', views.ArtistDetailView.as_view(), name='artists-detail'),
    path('entries/', views.DataListView.as_view(), name='data-overview'),
    path('entries/<str:pk>/', views.DataDetailView.as_view(), name='data-detail'),
    path('entries-table-adv/', views.data_table_advanced, name='data-table-adv'),
    path('entries-table-adv/export_csv', views.csv_export_adv, name='csv-export-adv'),
    path('entries-table-simple/', views.data_table_simple, name='data-table-simple'),
    path('entries-table-simple/export-csv', views.csv_export_simple, name='csv-export-simple'),
    path('museums/', views.MuseumListView.as_view(), name='museums-overview'),
    path('museums/<int:pk>/', views.MuseumDetailView.as_view(), name='museums-detail'),
    path('db_info/', views.db_info, name='database-info'),
    path('city-of-execution/', views.city_of_execution_overview, name='city-of-execution-overview'),
    path('city-of-execution/<str:pk>/', views.city_of_execution_detail.as_view(), name='city-of-execution-detail'),
]
```

Fig. 3: dttg_new/data/urls.py. Lists all the available URL's a user can request.

The above image shows what needs to happen when a user requests a certain URL. For example, in the case of the home. The user goes to webpage *https://domain.com/* , this file tells Django to redirect to the function `views.about` and the webpage has a name called `'dttg-home'`.

`views.about` refers to another python file: `dttg_new/data/views.py`. This python file basically dictates what should happen when a user requests a certain webpage.

So, at this point we have the following trajectory:

A user requests a URL →
Django looks up this URL in the `urls.py` file →
redirects to a specific view in the `views.py` file.

`views.py`

All the URLs in the `url.py` files, refer to a specific view function within the `views.py` file.

In its essence, a view can be a simple request-function (request because the user requests something), that may look like this:

```
30  def home(request):
31      return render(request, 'data/dttg-home.html')
```

Fig. 4: `dttg_new/data/views.py`. Function that describes what should happen when the user requests the `/home/` URL (in `urls.py`)

The function simply says: when this URL is being requested, render this request out using the following file: `'data/dttg-home.html'`. This file is called a template, and can be found in `dttg_new/data/templates/data/dttg-home.html`. As you may notice, this refers not to a .py file anymore, but a .html file: we've now moved on from the backend to the frontend.

Other views in `views.py` are more complex, but shall not be discussed here. Besides the `request` and `template-file` arguments in the function, a third argument `'context'` can be inserted, which allows to feed specific information (E.g., data, variables etc.) to the HTML file that can then be used from within there.

So, at this point we have the following trajectory:

A user requests a URL →
Django looks up this URL in the `urls.py` file →
Redirects to a specific view in the `views.py` file →
Redirects to a specific HTML-template in the `templates/data` folder

Frontend: [HTML, CSS, JS within the Django project]

Intro

The frontend (lay-out, user interface etc.) is not created with python, but instead using a combination of HTML, CSS, and JS. HTML (HyperText Markup Language) is the standard markup language for web pages. CSS (Cascading Style Sheet) is the language used to style an HTML document, as it describes how HTML elements should be displayed. Lastly, JS (JavaScript) allows for using certain scripts within HTML pages.

In `dttg_new/data/static/data/main.css` the main lay-out classes and colours are defined. Changes made in this file (for example a hex-code) will therefore translate to consistent changes all throughout the website.

The HTML templates in `dttg_new/data/templates/data/` describe exactly what text, what data, what files etc. should be presented to the user in what fashion/lay-out.
`dttg_new/data/templates/data/base.html` is the basis for all these pages, and describes all features that are constant across all pages. This includes the navigation bar at the top, the colour of the background, the footer etc. The other HTMLs therefore usually start with saying `{% extends 'data/base.html' %}`

Without going into the details of HTML coding, if something needs to be changed on one of the pages (E.g., an explanation), it shouldn't be too hard to find the corresponding text in the .html file where you can simply edit the text!

User interface/website

The list of pages that the user can access is the following:

- Informational pages
    - dttg-home.html (Home page that welcomes the user, and should explain the functionality of the website/database)
    - dttg-about.html (Basically copied the NICAS project-page information, added team-members and institutions etc.)
    - db-info.html (Info on the colour system, reliability system, and data collection)
- Artists
    - artist-list.html (lists all artists as cards)
    - artist-detail.html (detailed page of a single artist)
- Museums
    - museum_list.html (lists all museums as cards)
    - museum_detail.html (detailed page of a single museum)
- City of execution
    - city-of-execution-overview.html (lists all cities of execution)
    - city_detail.html (detailed page of a single city of execution)
- Data entries/Artworks
    - data_list.html (lists all artworks in the database)
    - data_detail.html (detailed page of a single artwork)
    - table_simple_query.html (table view of data, with simple search function)
    - table_advanced_query.html (table view of data, with advanced search function)

Find an overview in video-format here.

The built website/frontend thus basically allows the user to quickly glance at entries sorted on artist, city of execution, and museum, as well as provides them with a more elaborate table view in which more complex queries can be asked. These can then be exported as .csv to (for example) enter into more sophisticated data-visualisation tools.

Using the System Locally

To boot up the system locally, a couple things are required to set up:

- A terminal;
- An easy-to-navigate text-editor;
- Python (3.9.10);
- Virtual environment;
- Python extensions, including Django.

Terminal

On windows, it is recommended to download a terminal from an external party, as the built-in windows terminal is not as advanced. The one I recommend is Git BASH.

On mac, the internal terminal suffices.

Text-editor

Any text editor will work for opening and editing the different .py and .html files. However, native text editors as Notepad etc. do not have any automatic formatting options, which is very useful for interpreting and editing code. Therefore, I recommend installing SublimeText. This text-editor allows you to open the entire file tree, and also automatically colours certain parts of code, making it more legible.

Python

Python is the basis of the entire project. I recommend downloading the same version as I used, to prevent any compatibility issues: Python 3.9.10 .

Virtual environment

Instead of installing the required Python extensions on your computer, it is better practice to create a "virtual environment". This allows for working project-based, keeping different projects separated such that extensions don't start interfering with one another.

To do this, follow these instructions:

MAC	WINDOWS

In your terminal, write the following to install the virtual environment package:

MAC	WINDOWS
`python3 -m pip install virtualenv`	`pip install virtualenv`

Now, navigate to the folder in which you want to save your virtual environment, using the `cd` (change directory) command. E.g. `cd Desktop`. Then, create a virtual environment using the following command.

MAC	WINDOWS
`python3 -m virtualenv` *`NAME_OF_YOUR_ENVIRONMENT`*	`Python -m virtualenv` *`NAME_OF_YOUR_ENVIRONMENT`*

Stay in the same directory. To boot the venv up, use the following command:

MAC	WINDOWS
`source NAME/bin/activate`	`source NAME/Scripts/activate`

In your terminal, the venv should now appear in between closed brackets, indicating the boot up was successful!

Python extensions

The `requirements.txt` file indicates what python packages are required to run the Django project. These are:

- `Django`
- `Pillow`
- `django-crispy-forms`
- `django-extensions`
- `django-filter`
- `django-htmx`

To install these packages, use the `pip install` function from within your terminal (while your venv is running). On Mac that is: `python3 -m pip install PACKAGE_NAME`; On windows: `pip install PACKAGE_NAME`. Use the package names capitalized as above.

When done, type `python3 pip list` (Mac) or `pip list` (Windows) to check whether they are installed properly. This will also list their versions.

Boot-up

Now, it' s time to boot up our system!

Starting from scratch, the steps are:

1. Open up the project in *SublimeText* by dragging the folder into the software.
2. Boot up your terminal.
3. Navigate to the virtual environment and boot it up
4. Navigate to the project folder
5. Type python3 manage.py runserver (Mac) or python manage.py runserver (Windows)
6. This will now boot up the system on a local server. In the terminal you will see you IP-address (e.g. 1.270.0.0.1:8000). The part behind the colon is the so-called 'port-number'. You need this to access the webpage.
7. Go to any browser and type in localhost:*port_number* (e.g. localhost:8000). This will open up the webpage for you.
8. To stop running the server, which is recommended before you close the terminal, press Ctrl+C.
9. DONE!