A Guide into Preflib 2.0

Simon Rey

October 6, 2021

Contents

Chapter 1

Developing PrefLib

In this chapter, we detail the inner structure of PrefLib. We will first focus on the folder structure, explaining the role of each file. The second part of this chapter is devoted to the structure of the database.

The aim of this chapter is to provide all the necessary information to someone who would like to develop further the website. If you are only interested in maintaining the website and do some small changes, the second chapter might be more interesting to you.

1.1 Folder Structure

The folder structure of the project follows that of a typical Django project with one application. The overall Django project is called `preflib` and the main, and only, django application is called `preflibApp`. The overall folder structure can be found in Figure 1.1. In what follows, we give an explanation for each folder and files.

We start by detailing the `preflib` folder.

🗀 preflib

The `preflib` folder contains the files that have an impact on the entire project. This is the highest folder in the Django hierarchy. Files in there are mainly used to set up the global parameters of the project.

🗋 **settings.py** This is the main files for the global settings. Among other things, you will find there the settings for the database, the location of the static files, the debugging mode, the installed applications, ... Not that this file is not on the git for security reasons.

🗋 **urls.py** Use this file to set up the global rules for urls. Whenever a request passed over to Django, this file is used to decide where to send the request next. Handlers for the errors (404, 500, etc...) are also defined there.

🗋 **wsgi.py** This is only used to set up the connection between Django and whatever WSGI tool is used (uWSGI, Gunicorn, Passenger). If you did not understand the previous sentence, you will most likely never have to deal with this file.

Let us now move to the `preflibApp` folder.

```
preflib
    settings.py
    urls.py
    wsgi.py
preflibApp
    management
        commands
            adddataset.py
            generatezip.py
            initializedb.py
            updatemetadata.py
            updatepapers.py
    preflibtools
        drawing.py
        graph.py
        instance.py
        properties.py
    static
        adminLogo
            images for the admin menu
        css
            nouislider.css
            style.css
        data
            CD
            ED
            MD
            types
        datatoadd
        js
            nouislider.js
        types
            dataTypesImages.tex
            images to present the types
        crisner.png
        gitHubLogo.png
        papers.bib
        preflibADTCite.txt
        preflibADTPaper.pdf
        preflibTrendsComSoCChapter.pdf
        preflibTrendsComSoCCite.txt
    templates
        includes
            HTML files that can be included
        preflib
            main HTML files
        400.html
        403.html
        404.html
        500.html
    templatetags
        extratags.py
    admin.py
    apps.py
    choices.py
    forms.py
    models.py
    scripts.py
    urls.py
    views.py
```

Figure 1.1: Folder Structure of the Project, some less relevant files have been omitted.

preflibApp

This folder contains all the files related to the `preflibApp` Django application. Because this is the only application we have, it thus contains the entire website. Let us first go through the Python files you can find there, sub-folders will be examined afterwards.

urls.py This file sets the url patterns for the application. It is in there that the link between an URL and the corresponding view is made. It is also in there that the set of all acceptable URLs (that will not return a 404) is defined. When a request is passed over to Django, the URL is first filtered by the `urls.py` file in the `preflib` folder which then calls the url pattern defined in this file to know what view to call for.

views.py In Django, a view is a function that takes an input a request and that return a rendered template (an HTML file). This files gathered all the views. This file is probably the most important one as it includes all the code that is run once a request arrives: updating the database, computing so stuff, filter the entries of the database, etc.

forms.py Django offers a Python class to deal with forms that makes it easier to check whether they have been properly filled in etc... The forms are defined in this file. It, for instance, includes the search form, the login form, etc.

models.py A model in Django is the corresponding Python class to a table in the database. In this files all the models are described in Python. Django then reads through this file and create the database accordingly. The entire database is thus defined in there.

choices.py Choices are lists of constant values that are not meant to change often (they would be in the database otherwise). An example is the set of data category for instance. This files gathered them all.

admin.py In this file, one can register the models so that they appear in the Django admin website.

scripts.py This file defines few useful functions for management purposes.

apps.py This is a file that is used by Django to know the application exists. Do not modify it.

For the rest of this section, we will explore the sub-folders present in the `preflibApp` folder.

preflibApp/templates

In Django, a template is an HTML that can incorporate some Django tags in it. They are the main files containing the HTML code of the website. They all are gathered in the `templates` folder. It contains all the template to render the errors: `400.html`, `404.html`, etc. It also contains several sub-folders used to separate the templates based on their use.

includes This folder gathers all the templates that can be included in some other templates using the {% `include 'template'` %} Django tag. These files are:

- footer.html: the footer displayed at the bottom of a page;
- header.html: the header displayed at the top of a page;
- htmlHeaderContent.html: the content of the HTML header that is shared with all pages;
- metadataCategorySearch.html: the code used to render a search widget in the search page;
- paginator.html: the code used to render a paginator in a page.

🗀 **preflib** This folder gathers the templates used to render all the pages of the website. We list them below.

- 🗋 about.html: the about page
- 🗋 admin.html: the admin menu
- 🗋 adminadddataset.html: the admin page for adding datasets
- 🗋 adminlog.html: the admin page for viewing logs
- 🗋 adminpaper.html: the admin page for adding a paper
- 🗋 adminzip.html: the amdin page for zipping the files
- 🗋 data.html: the data page describing the structure of the data
- 🗋 dataformat.html: the page explaining the format we use
- 🗋 datametadata.html: the page describing the metadata
- 🗋 datapatch.html: the template for rendering a datapatch
- 🗋 datasearch.html: the search page
- 🗋 dataset.html: the template for rendering a dataset
- 🗋 datasetall.html: the template for rendering a category of data
- 🗋 datatypes.html: the page describing the types
- 🗋 index.html: the main page of the website
- 🗋 toolscris.html: the Crisner tool page
- 🗋 toolsivs.html: the Iterative Voting Simulator page
- 🗋 toolsskdg.html: the Kidney Dataset Generator page
- 🗋 userlogin.html: the login page
- 🗋 usernewaccount.html: the page to create new account
- 🗋 userprofile.html: the page displaying the profile of a user

🗀 preflibApp/templatetags

The `templatetags` folder is used to describe user-defined tags that can then be used in the templates. It contains a single file—`extratags.py`—where the extra tags are defined.

🗀 preflibApp/static

The `static` folder contains all the static files that are used (unsurprisingly). This folder is the one considered by the `collecstatic` management command from Django.

🗀 **adminLogo** This folder gathers all the images of the logo used for the admin menu. Note that it might disappear in some later versions.

🗀 **css** This folder gathers all the CSS files. There are currently two of them: `style.css` which is the main CSS file for the style of the website and `nouislider.css` that is used to render the sliders in the search page.

🗀 **js** This folder gathers all the JavaScript files. The only file in there is used for the sliders in the search page.

🗀 **types** This folder gathers the images used to present the types together with the TeX file generating some of them.

🗋 **crisner.png** This image illustrates the Crisner tool.

🗋 **gitHubLogo.png** This image is the GitHub logo displayed in the footer.

🗋 **papers.bib** This bib file is the one read by the management command `updatepapers`. It contains all the bib entries that are then put in the database as "papers using PrefLib".

🗐 **preflib citations** For the ADT paper and the Trends in ComSoC chapter, the pdf and a txt file with the bib information are provided.

🗀 preflibApp/management/commands

The `management/commands` folder includes the user-defined commands that one can access using the Django's management tool. Each file in this folder contains a class `Command` which should contain some specific methods. Importantly, once defined and put in this folder, the command can be accessed as any other management command, for instance from the command line using: `python3 manage.py adddataset`.

🗋 **adddataset.py** This command is used to add datasets to the database. Only zip files located in the folder `static/datatoadd` can be retrieved by this command. Two arguments can be passed to the command, either `--file zipfilename` to add only a specific zip file, or `--all` to add all the zip files in the `static/datatoadd` folder to the database.

When adding a dataset, the command unzip the archive in a temporary folder. It then goes through the files to find the info file used to get all the information for the dataset. Then, each file is added to the database, together with its datapatch (if needed). Note that the metadata are not computed when adding a dataset.

🗋 **generatezip.py** This command is used to generate all the zip files served by the website. It first generates the zip files for the dataset, creating the info file with the entry in the database. In a second time, it generates the zip files per type of data.

🗋 **initializedb.py** This command is only run once, at the very beginning, to populate the database with the entries that are needed. It mainly sets up the metadata.

🗋 **updatemetada.py** This command computes the metadata for the data files. Two options can be passed to the command. When used with `--dataset datasetAbbreviation`, only the metadata for the given dataset will be computed. One can also use `--noDrawing` to avoid generating the images for the data file (which takes a lot of time).

For each relevant data file, the command will go through all the *active* metadata (see later what active mean here). Whenever an active metadata applies to the data type of the data file, the corresponding function is called to compute its value. If drawing is allowed, the relevant drawing method is also called.

🗋 **updatepapers.py** This command updates the list of papers that are using Preflib. It reads the bib file `static/papers.bib` and update the database accordingly.

🗀 preflibApp/preflibtools

This folder contains the tools that were developed around PrefLib to work with the data. Note that this version is far from what you can find in the PrefLibTools repository on GitHub. These tools are mainly used to analyze the data when computing the relevant metadata.

🗋 **drawing.py** This file contains all the functions that are used to draw the images representing the data.

graph.py This file contains the definition of the `Graph` class that we use to represent graphs and access several useful methods for them.

instance.py This file contains the class used to represent PrefLib instances. The methods for parsing the data file are defined there.

properties.py This file contains a collection of functions used to check whether some properties hold for a given instance. This is the main file used to compute the metadata of the instances.

1.2 Database Structure

We now turn to the database used behind the website to organize the datafiles. In igure 1.2 you will find a complete schematic representation of the database. In what follows, we will give more explanations for each table and row.

DATAFILE

The most fundamental entity for Preflib is the datafile. The DATAFILE table contains a reference to all the datafiles that are in Preflib. The table does not contain the data in itself—it is stored in a file and not in the database—but all the relevant information about it: some basic details and datapatch in which the file is.

In what follows, we provide more details on each of the columns of this table:

- **dataPatch**: this is a foreign key on a datapatch entry. It indicates the datapatch in which the datafile is.
- **dataType**: the type of the datafile, it should be one of soc, soi, toc, toi, tog, mjg, wmg, pwg, wmd, dat, csv. This field is linked to the `DATATYPES` choice list to get a consistent database.
- **modificationType**: the modification type of the datafile, it should be one of original, induced, imbued, synthetic. This is linked to the `MODIFICATIONTYPES` choice list to get a consistent database.
- **fileName**: the name of the file containing the actual data.
- **fileSize**: the size of the file containing the actual data.
- **publicationDate**: the date at which the file has been added to the database.
- **modificationDate**: the date at which the file has been last modified.

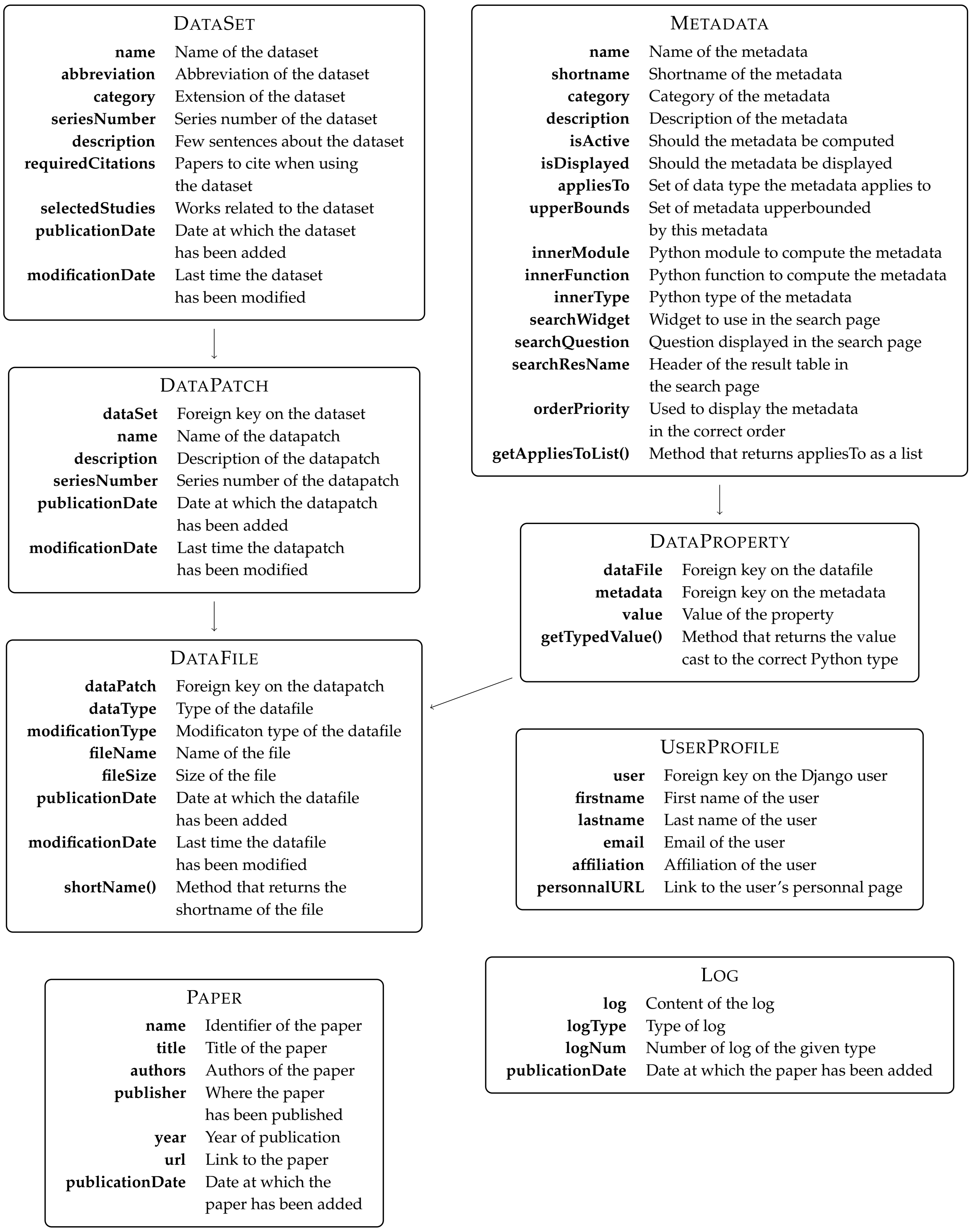
The Python class also introduces a method called `shortName()` which can be used to get the shortname of a datafile. This method is used for the name of the datafiles in the result table of the search page. Note that this is a method of the Python class and not a column of the table in the database.

DATAPATCH

The datapatch is the first level of classification of the datafile. It contains several datafile of different datatype. All the datafiles are based on the same preferences but the representation, the datatype, is different.

In what follows, we provide more details on each of the columns of this table:

- **dataSet**: this is a foreign key on a dataset entry. It indicates the dataset in which the datapatch is.
- **name**: the name of the datapatch.
- **description**: a short description of the datapatch. It usually indicates the exact content of the patch with respect to the full dataset. For instance in the Irish elections dataset, it could be “Dublin 2006”.
- **seriesNumber**: the identifier of the datapatch, this is a 8 digits number.
- **publicationDate**: the date at which the datapatch has been added to the database.
- **modificationDate**: the date at which the datapatch has been last modified.



DATASET

name	Name of the dataset
abbreviation	Abbreviation of the dataset
category	Extension of the dataset
seriesNumber	Series number of the dataset
description	Few sentences about the dataset
requiredCitations	Papers to cite when using the dataset
selectedStudies	Works related to the dataset
publicationDate	Date at which the dataset has been added
modificationDate	Last time the dataset has been modified

DATAPATCH

dataSet	Foreign key on the dataset
name	Name of the datapatch
description	Description of the datapatch
seriesNumber	Series number of the datapatch
publicationDate	Date at which the datapatch has been added
modificationDate	Last time the datapatch has been modified

DATAFILE

dataPatch	Foreign key on the datapatch
dataType	Type of the datafile
modificationType	Modificaton type of the datafile
fileName	Name of the file
fileSize	Size of the file
publicationDate	Date at which the datafile has been added
modificationDate	Last time the datafile has been modified
shortName()	Method that returns the shortname of the file

PAPER

name	Identifier of the paper
title	Title of the paper
authors	Authors of the paper
publisher	Where the paper has been published
year	Year of publication
url	Link to the paper
publicationDate	Date at which the paper has been added

METADATA

name	Name of the metadata
shortname	Shortname of the metadata
category	Category of the metadata
description	Description of the metadata
isActive	Should the metadata be computed
isDisplayed	Should the metadata be displayed
appliesTo	Set of data type the metadata applies to
upperBounds	Set of metadata upperbounded by this metadata
innerModule	Python module to compute the metadata
innerFunction	Python function to compute the metadata
innerType	Python type of the metadata
searchWidget	Widget to use in the search page
searchQuestion	Question displayed in the search page
searchResName	Header of the result table in the search page
orderPriority	Used to display the metadata in the correct order
getAppliesToList()	Method that returns appliesTo as a list

DATAPROPERTY

dataFile	Foreign key on the datafile
metadata	Foreign key on the metadata
value	Value of the property
getTypedValue()	Method that returns the value cast to the correct Python type

USERPROFILE

user	Foreign key on the Django user
firstname	First name of the user
lastname	Last name of the user
email	Email of the user
affiliation	Affiliation of the user
personnalURL	Link to the user's personnal page

LOG

log	Content of the log
logType	Type of log
logNum	Number of log of the given type
publicationDate	Date at which the paper has been added

Figure 1.2: Structure of the database

DataSet

A dataset is a collection of datapatches. The datapatches will typically represent different years of the same election, all gathered in a unique dataset.

In what follows, we provide more details on each of the columns of this table:

- **name**: the name of the dataset
- **abbreviation**: the abbreviation of the dataset. That will be the name of the zip archive containing the dataset for instance.
- **category**: the category of the dataset, it should be one of ED, MD, CD. This field is linked to the `DATACATEGORY` choice list to get a consistent database.
- **seriesNumber**: the identifier of the dataset, this is a five digit number.
- **description**: a brief description of the dataset, usually a paragraph. It is rendered as HTML code so any HTML tags can be used there.
- **requiredCitations**: a list of papers that are to be cited if this dataset is used in a publication.
- **selectedStudies**: a set of papers that are relevant for this dataset, typically showing examples of what have been done with it.
- **publicationDate**: the date at which the dataset has been added to the database.
- **modificationDate**: the date at which the dataset has been last modified.

DataProperty

A dataproperty is an additional information about a given datafile. It can have to do about the general properties of the data (number of candidates...) or about some more specific structure of the data (single-peakedness...).

In what follows, we provide more details on each of the columns of this table:

- **dataFile**: a foreign key on a datafile entry. It indicates the datafile the dataproperty is about.
- **metadata**: a foreign key on a metadata entry. It indicates the type of metadata the dataproperty is about.
- **value**: the actual value of the metadata for the given datafile. It can be of many types, most likely it will be either a number or a boolean.

The Python class also introduces the method `getTypedValue()` which returns the value of the dataproperty but cast in the correct Python type (it is otherwise a string). This method is mainly used to filter the datafiles in the search tool. Note that this is a method of the Python class and not a column of the table in the database.

Metadata

A metadata is a type of information about a datafile. The actual value of the datafile for a given metadata is stored in DataProperty table.

In what follows, we provide more details on each of the columns of this table:

- **name**: the name of the metadata
- **shortname**: the shortname of the metadata, it should not contain any space character. It is used in the internal of the search tool.
- **category**: the category of the metadata, it should be one of general, preference, ballot. This filed is linked to the `METADATACATEGORIES` choice list to get a consistent database.

- **description**: the description of the metadata as displayed in the metadata page.
- **isActive**: a boolean value indicating whether the metadata should be considered when computing the dataproperties through the `updatemetadata` management command.
- **isDisplayed**: a boolean value indicating whether the metadata should be displayed in the website. Some metadata can be interesting for sanity checks but not as a search field for instance.
- **appliesTo**: a string representing all the type of data the metadata applies to. The types should be separated by a coma "," and without spaces, e.g., "toi,toc,soi,toc"
- **upperBounds**: a many-to-many relationship on metadata entries. Any metadata in the many-to-many set is upper-bounded by the current metadata. It of course only applies to metadata with numeric values. It is used in the search page to synchronize the sliders of some metadata.
- **innerModule**: the internal Python module used to compute the value of the metadata. This is how the `updatemetadata` command knows in which module the function to call to create a dataproperty entry is.
- **innerFunction**: the internal Python function used to compute the value of the metadata. This is how the `updatemetadata` command knows which function to call to create a dataproperty entry.
- **innerType**: the internal Python type of the metadata. That type is used to cast the value correctly when using the `getTypedValue()` method of a dataproperty object.
- **searchWidget**: the widget used to display the corresponding field in the search page form. It should be one of ternary, range. This filed is linked to the `SEARCHWIDGETS` choice list to get a consistent database.
- **searchQuestion**: the question displayed together with the widget on the search page.
- **searchResName**: the header of the corresponding column used in the result table of the search page.
- **orderPriority**: a integer value that is used to order the metadata in a fixed way. This is used to fix the order of the metadata page for instance.

The Python class also defined the `getAppliesToList()` method which returns the appliesTo field but as a list of string instead of just one full string. Note that this is a method of the Python class and not a column of the table in the database.

PAPER

This table stores the information about the papers that are using Preflib displayed on the front page.

In what follows, we provide more details on each of the columns of this table:

- **name**: identifier of the paper, typically the bib identifier in the bib file.
- **title**: the title of the paper.
- **authors**: the authors of the paper, just the initial of the first name, without a dot, and the family name afterwards.
- **publisher**: the conference, journal, venue... in which the paper has bee published.
- **year**: the year of publication of the paper.
- **url**: a URL linking the to paper.
- **publicationDate**: the data at which the paper has been added to the database.

UserProfile

All the information about the users that are not in the Django User class are present in this table. Note that for the moment this is not used.

In what follows, we provide more details on each of the columns of this table:

- **user**: a foreign key on the Django user entry.
- **firstname**: the first name of the user.
- **lastname**: the last name the user.
- **email**: the email of the user.
- **affiliation**: the affiliation of the user.
- **personnalURL**: a URL linking to the website of the user.

Log

Logs of what is happening in the inside are gathers in this table.

In what follows, we provide more details on each of the columns of this table:

- **log**: the actual content of the log. It is rendered as HTML code so any tag in there will be rendered accordingly.
- **logType**: the type of the log.
- **logNum**: the identifier of the log inside its category of log.

Chapter 2

Maintaining PrefLib

We now turn to the more practical ways of maintaining PrefLib. The chapter is organized in terms of actions, each corresponding section explaining how to perform the given action. Reading the previous chapter will be helpful to understand what is going on in what follows, but not required.

Before moving to some specific actions, let me discuss some generalities about the current implementation on the server. Whenever an HTTP request is sent to the server, it is received by the Apache web-server. This server is configured to listen to some specific ports and to serve the request by matching the requested URL to the corresponding rule. Theses rules are defined on the DreamHost settings and will likely never be changed (or the person doing it will know what she's doing). For PrefLib, the request is sent to a second web-server, called Passenger, that makes the link between the Apache server and the Django project. Passenger is also configured via DreamHost, and most likely will not need maintenance. Finally, Passenger sends the request to the Django project that returns an HTTP response defined by the corresponding view.

One important thing to keep in mind is that Passenger is the tool that launches the Django project. Therefore, whenever some changes have been made to the Django project, one needs to restart Passenger to take this changes into account. This is done by updating the last modification date (using the command `touch`) of the file `tmp/restart.txt`. So assuming some changes have been made and that one has a terminal opened in the suitable folder, one should run:

```
touch tmp/restart.txt
```

Inside the folder corresponding to the website, there is a folder called `PrefLib-Django`. This folder is where the Django project is located. This folder is linked to the git repository and so pulling from the git should be done within this project. One important file there is `manage.py` which is a python script used to run all maintenance commands for the Django project. We will see several usage of that file in what follows.

To keep things clean, all the required Python library are installed in a virtual environment. What that means is that to run any of the commands mentioned in this chapter, one has to activate the virtual environment first. To do so, run the following command from the `PrefLib-Django` folder.

```
source ../preflibvenv/bin/activate
```

The prompt of the terminal should change, to indicate the virtual environment is active. Once, you are done, exist the virtual environment with the following command (from any folder).

```
deactivate
```

2.1 Creating a Super-User

A super user is a user that has all the rights, to create one simply run the following command.

```
python3 manage.py createsuperuser
```

2.2 Updating Static Files

The static files (images, datafile, etc...) are served differently than other requests. To do so, Django collects all the static files of the projects in a specific folder (whose location is defined in the `preflib/settings.py` file). If a static file has been modified, or if a new one has been added, it will be served only if it is present in the static folder (not the one from the Django project, but the one defined in `preflib/settings.py`). To copy the static files into the correct folder, run the following command (from the `PrefLib-Django` folder).

```
python3 manage.py collectstatic
```

2.3 Updating HTML Files

As explained in the previous chapter, HTML filed are called "templates" in Django and can be found in the folder `preflibApp/templates`. To update the HTML code of a page, simply edit the corresponding template file. Please note that this files are HTML files, but also contain Django language. Once your are done editing the files, do not forget to restart Passenger.

2.4 Modifying the Database Structure

The structure of the database is described in the file `preflibApp/models.py`. If you want to make changes to the database (to add a field or a table for instance), simply make those changes in the file `preflibApp/models.py`. Once you are done with the edits, you need to run two commands. First, run the command

```
python3 manage.py makemigrations
```

that will check for changes in the models are create the corresponding Python script to update the database. Second, run the command

```
python3 manage.py migrate
```

that will actually apply the migrations and make the changes to the database.

2.5 Initializing the Database

When the website is installed for the first time, one has to initialize the database, that is done by running the following command:

```
python3 manage.py initializedb
```

2.6 Adding Dataset(s)

To add a dataset to the website, one has to have it in a zip archive already formatted in the PrefLib format. In particular in this archive there should be all the datafile and an `info.txt` file describing the datafile. The easiest to see what is expected is to download on of the zip file of a dataset from the website and see what is in there (here for instance).

To add a unique dataset, one can run the command:

```
python3 manage.py adddataset --file zipFilePath
```

where `zipFilePath` is the path to find the zip file of the new dataset.

There is also a way to add several datasets in one go by running the command:

```
python3 manage.py adddataset --all
```

in which case all the zip files in the folder `preflibApp/static/datatoadd` are added to the website.

Note that static are automatically collected at the end of the procedure, no need to do it manually. However several actions have to be performed after adding a dataset: generating the zip files and updating the metadata. These are described below.

2.7 Generate Zip Files

The website serves some pre-compiled zip files, namely the zip files of each datasets and the zip files for each datatype. To re-generate these files, run the command:

```
python3 manage.py generatezip
```

This command has to be run after adding a dataset for instance, the static files are however automatically collected.

2.8 Updating the Metadata

The new website collects a bunch of metadata about the datafile, those are computed by running a specific command. This also is the command used to generate the visualization of the data. The command can be run over all datasets, using the following:

```
python3 manage.py updatemetadata
```

It can also be run for a specific dataset by running:

```
python3 manage.py updatemetadata --dataset datasetAbbreviation
```

where `datasetAbbreviation` is the abbreviation (such as `irish` or `f1`) of a dataset.

This command can take a lot of time (several hours), it is thus recommended to run it in a `tmux` environment. The most time consuming task at the moment is the generation of the images for the datafiles, this task can be disable by passing the argument `--noDrawing` to the command.

2.9 Updating the List of Papers Using PrefLib

We maintain a list of papers that are using PrefLib (displayed on the front page of the website). The entries in the database are read from the bib file `preflibApp/static/papers.bib`. To match the entries in the database to the bib file, run the following command:

```
python3 manage.py updatepapers
```

2.10 A Comprehensive Example

Let us now go through a comprehensive example to show how things work. Consider that you have a terminal opened, already on the server and in the folder `preflib.org`.

The first thing to do is to activate the virtual environment

```
source preflibvenv/bin/activate
```

Then, let's move to the Django project folder

```
cd PrefLib-Django/
```

Assume the website has just been set up. We then initialize the database

```
python3 manange.py initializedb
```

We also create a superuser that will have access to the admin website

```
python3 manage.py createsuperuser
```

We create the entries in the database for the papers that are using PrefLib

```
python3 manage.py updatepapers
```

We now copy all the datasets we want to add to the website in the folder `preflibApp/static/datatoadd`. We check with just one that everything is working

```
python3 manage.py adddataset --file preflibApp/static/datatoadd irish.zip
```

Everything worked, so we add them all

```
python3 manage.py adddataset --all
```

Now that the datasets have been added, we compute the metadata for them. We first check that it is working for one dataset

```
python3 manage.py updatemetadata --dataset irish
```

Everything worked so we do it for all datasets

```
python3 manage.py updatemetadata
```

We also generate the static zip files that are served

```
python3 manage.py generatezip
```

The website is now good to go, we deactivate the virtual environment

```
deactivate
```

We are now done.