**PART 1: SETTING UP THE PROJECT**

1. Create your project folder and move inside it. I called mine films\_project.

mkdir films\_project  
cd films\_project

2. Create a new virtual environment with venv (mine is called .myenv).

python -m venv .myenv

3. Now activate your virtual environment. The command below works on the GitBash. Consult this [Python documentation page](https://docs.python.org/3/library/venv.html) to check the command for both your terminal and OS.

source .myenv/Scripts/activate

4. Install the Python packages we will need for this project:

pip install django django-extensions

5. Create a requirements.txt file. You can use the following command:

pip freeze | grep -i django >> requirements.txt

6. Start a new Django project. Mine is just called project. Don’t forget to add the dot at the end of this command.

django-admin startproject project .

7. Start a Django app called films.

python manage.py startapp films

8. Open project/settings.py and add django\_extensions and the films app to the INSTALLED APPS list . Don’t change the other lines.

INSTALLED\_APPS = [  
 'django.contrib.admin',  
 'django.contrib.auth',  
 'django.contrib.contenttypes',  
 'django.contrib.sessions',  
 'django.contrib.messages',  
 'django.contrib.staticfiles', # add these lines:   
 'django\_extensions',  
 'films.apps.FilmsConfig',  
]

9. Apply the migrations to create the tables in the sqlite3 database.

python manage.py migrate

10. Create a superuser by running the following command:

python manage.py createsuperuser

Inform an username, email, and password. You’ll use this data to login in the admin area later.

11. Run the server with python manage.py runserver , visit http://localhost:8000 and check if the app is working correctly.

12. Go to http://localhost:8000/admin , use your login data and check if you can access the admin area.

**PART 2: DEFINE THE MODELS**

13. In films/models.py, enter the following code:

We are creating two model classes(Film and Genre), with a simplified relationship between them, since one film can have just one genre here. In the future, you can change to a many-to-many relationship between these tables, if you want, or you can even add other entities to the model, like directors or awards.

In the Films class, notice the important get\_fields() method. This will create a list of label/value pairs for every column in the films\_film table but the id, a structure that we can use in a for loop inside the future film\_detail.html template. Such a structure could be even more helpful if we had dozens or even hundreds of columns in a model, so this code might be worth saving for future projects.

**PART 3: CREATE NEW CATEGORIES USING THE ADMIN AREA**

14. Run the commands to create the new tables in the database:

python manage.py makemigrations  
python manage.py migrate

15. Open films/admin.py and write these lines. Now the Films and Genre models will become available in the admin area:

from django.contrib import admin  
from .models import Film, Genreadmin.site.register(Film)  
admin.site.register(Genre)

16. Next, login to the admin area, select the Genre model, then create and save two or three new genres. We will need these genres later, when we use the films CRUD to save new movies.

**PART 4: STRUCTURE THE APP ROUTES AND CLASS-BASED VIEWS**

17. Create the films/urls.py file.

18. Open project/urls.py , import the include function and use it to create a new route:

from django.contrib import admin  
from django.urls import path, include # addedurlpatterns = [  
 path('', include('films.urls')), # added  
 path('admin/', admin.site.urls),  
]

19. Next, open films/urls.py and add the following code to it.

The CBV are not created yet, so running the server at this point will generate errors.

20. In films/views.py, this code will create the generic CBV that we need:

Notice how concise this code is: we only create a FilmBaseView so that we don’t repeat the model, fields and success\_url info lines, then we make our Film View Classes inherit from both FilmBaseView and the respective Generic View Classes that Django already has for doing CRUD operations. All that with no need of working with forms logic, or writing code to access the database and then save this info in a context dictionary. Those procedures are already worked out by Django’s CBV. With the advantage that we can always modify any of their methods and add new behavior to the View Classes depending on our specific needs.

Now we only need to create the templates to display and change the information provided by our Film Class Views.

**PART 5: CREATE THE TEMPLATES**

21. Open project/settings.py and, in the variable TEMPLATES, fill the empty list in the TEMPLATES['DIR'] key with the following information (change only this one line, keeping the other untouched):

TEMPLATES = [  
{(...)'DIRS': [ BASE\_DIR.joinpath('templates') ],(...)}  
]

22. Make sure you are in the root project (folder films\_project, at the same level as manage.py), then create these three new folders.

mkdir templates films/templates films/templates/films

Now, it is time to create the HTML files we will use as templates to the films CRUD. However, before we do that, a pratical observation: as a matter of fact, I think that it will be better not to reproduce the templates code here, since most of it is too long. So, I will provide some links to the raw content from each template file saved on my GitHub repository, so that you can copy/paste or transcribe it to your own code.

23. Create a templates/base.html file. This file should not be inside the films template folder, be careful with that.

* touch templates/base.html ([content here](https://raw.githubusercontent.com/fabricius1/DjangoFilmsCRUD/master/templates/base.html));

24. Inside the films/templates/films folder, create four files with the exact following names, since those are the template names patterns used by Django’s generic Views:

* film\_confirm\_delete.html ([content here](https://raw.githubusercontent.com/fabricius1/DjangoFilmsCRUD/master/films/templates/films/film_confirm_delete.html));
* film\_detail.html ([content here](https://raw.githubusercontent.com/fabricius1/DjangoFilmsCRUD/master/films/templates/films/film_detail.html));
* film\_form.html ([content here](https://raw.githubusercontent.com/fabricius1/DjangoFilmsCRUD/master/films/templates/films/film_form.html));
* film\_list.html ([content here](https://github.com/fabricius1/DjangoFilmsCRUD/blob/master/films/templates/films/film_list.html));

You can create these files at once by running just this line of code in a Linux terminal (or Git Bash):

touch films/templates/films/film\_{confirm\_delete,detail,form,list}.html

Notice how these HTML file names are prepended with film\_ , which is our model name in smallcaps followed by an underscore. You should use this pattern when creating CRUD applications with CBV in future Django projects.

25. Run python manage.py check to see if there are any errors; then run the server with python manage.py runserver. Your CRUD films application is finished and now you can list, add, update and delete films information with it.

**PART 6: ADD SOME EXTRA FILMS TO YOUR DATABASE USING A PYTHON SCRIPT**

Explore your new app and make some changes on it. If you want extra data to play with, you can load to Django a list of more than 20 Pixar movies I prepared. You just need to follow these steps:

26. Create a scripts folder at the same level of manage.py

27. Create a films/pixar.csv file and [copy this content into it](https://raw.githubusercontent.com/fabricius1/DjangoFilmsCRUD/master/films/pixar.csv).

28. Create a scripts/load\_pixar.py file and [copy this content into it](https://raw.githubusercontent.com/fabricius1/DjangoFilmsCRUD/master/scripts/load_pixar.py).

29. Run python manage.py runscript load\_pixar. Access your films app and check how it is populated with the Pixar movies info now.