Django and Databases

*** Day 01 ***

We will work with the movie_theater project again. Your project must be saved here unit_3/movie_theater_project. Inside that folder set up a python environment, create the requirements files, and install the required packages. Please add the following packages in the dev.in file:

nodeenv
pgadmin4
pygraphviz
django-extensions

Build and Install packages:

pip install --upgrade pip-tools pip setuptools wheel
pip-compile --upgrade --generate-hashes --output-file requirements_env/main.txt requirements_env/main.in
pip-compile --upgrade --generate-hashes --output-file requirements_env/dev.txt requirements_env/dev.in

pip-sync requirements_env/main.txt requirements_env/dev.txt

Keep in mind in the production server we do not need those packages that is why we do not add them in the main.in file.

Create a blank project movie_theater:

django-admin startproject movie_theater

Let's run the server:

python manage.py runserver

Check if the website is working at: **localhost:8000**

By the way, do you see what django is telling us when we launch the runserver?

"""You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them."""

Now that we have access to a database. Our shared database at school, or on your computer at home (faster access and you will learn how to install and maintain a database). Other option at home is through a tunnel to access the school database (it is better to use the db on your own devices).

So to connect to a database we need to provide the host and credentials information.

Here we have some information on how to do it:
https://docs.djangoproject.com/en/4.2/ref/settings/#databases

We need to set up the host and login credential for the database into the file movie_theaterf/settings.py.

In this file, let's add the following imports before the line from pathlib import Path:

```
import os
import json
```

And add the following after the line pathlib import Path:

```
from django.core.exceptions import ImproperlyConfigured
```

Normally you should not import ANYTHING from Django directly into your settings, but ImproperlyConfigured is an exception, so that is fine.

We are importing an exception function from the django library.

After the BASE_DIR, let add this function:

```
# JSON-based secrets module
with open(os.path.join(
        BASE_DIR, movie_theater, 'secrets.json')) as f:
        secrets = json.loads(f.read())

def get_secret(setting, secrets=secrets):
        '''Get the secret variable or return explicit exception.'''
        try:
        return secrets[setting]
        except KeyError:
        error_msg = 'Set the {0} environment variable'.format(setting)
        raise ImproperlyConfigured(error_msg)
```

What is this function doing?
What do we need to make it?

Let's create the secrets.json file in the same folder as settings.py and let's put the host and login credentials we need for postgresql.

```
{
  "environment": "development",
```

```
  "movie_theater_url": "http://localhost:8000",
  "database_name": "database_name",
  "database_user": "username",
  "database_pwd": "pwd",
  "database_host": "149.89.160.100",
  "database_port": "5432"
}
```

Run the dev server, python manage.py runserver. If you have an error, figure it out.

The file secrets.json should **NEVER** be committed to the git repository. Never put any kind of credentials on a git repository, because in a git repository developers need access to the code but do not necessarily have access to the real data in a company. If you put the credentials in a git repository, anyone who works on the project will have access to the production data and this is a huge security breach.

So we save the credential in the secrets.json, but the Django project is not connected to the database yet, let's do this now.

In the setting.py file look for the variable DATABASES, and replace the content of that variable with the following code:

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql",
        "NAME": get_secret('database_name'),
        "USER": get_secret('database_user'),
        "PASSWORD": get_secret('database_pwd'),
        "HOST": get_secret('database_host'),
        "PORT": get_secret('database_port'),
    }
}
```

Check if your server is still working, if not try to find the solution for your issues.

Let's apply the migrations now on the database. Django has already built some models for us, and it will create some tables for us and we will be able to use them right away.

These are the commands you should run in your terminal to make the migrations:

python manage.py makemigrations => We do not need to do this right now, but let's get the habit. First thing you should always do is to check if you modify the models to apply those changes to the database. And for this, we need to run another command. If you did any modification that needed a change in the database, it is going to create a migration file inside

the migrations folder of the app concerned. You can read this page for more information: (https://docs.djangoproject.com/en/4.2/topics/migrations/)

This is the command to apply any migration files that has not been used by the database: **python manage.py migrate**

Right now, as our database is blank, it is going to create all the django default tables

So these are the migration that Django will apply to your DB;

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
```

Contentypes will register the app and the models that has been created in the database.

Auth tables will regroup the credentials that the user has with the application and also the permissions that can be set up by default with a django application.

Admin tables, will get you access to an admin interface already done for you. With a few lines in an admin.py inside the app folder, you can create an automatic interface to create and update data. We are not going to use it, but if you want to know more go see these urls https://docs.djangoproject.com/en/4.2/ref/contrib/admin/ and https://docs.djangoproject.com/en/4.2/intro/tutorial02/)

We also have a sessions table, where user session data will be stored there: Yay no more cookies!!!! :) Session and database will store the data we will need.

Go checkout your database with pgadmin, you will see there are new tables.

*** Day 02 ***

Superuser

Create a superuser on your Django project. This will be used to log in to the admin page, and to log in to your website when we will be working with authentication.

python manage.py createsuperuser

You will be asked to set up a username and password for that superuser.

Create two applications: movies and theaters.

python manage.py startapp movies
python manage.py startapp theaters

Models

A **Django model** is the built-in feature that Django uses to create tables, their fields, and various constraints.

Each model is a Python class that subclasses django.db.models.Model.

Example:

```
class Movie(models.Model):
      # PK id created by default id (bigint)
      # The following line will override the default bigint id.
      # The id will be int.
      # id = models.AutoField(primary_key=True)
      name = models.CharField(max_length=50, unique=True)
      running_time = models.TimeField()
      actors = models.ManyToManyField(Actor)
      director = models.CharField(max_length=200)
      release_date = models.DateTimeField()

class Screen(models.Model):
      # we cannot delete a theater if there is a screen associated to it
      theater = models.ForeignKey(Theater, on_delete=models.PROTECT)
      name = models.CharField(max_length=30)
```

Field Choices

Each field takes a certain set of field-specific arguments. See here:
https://docs.djangoproject.com/en/5.0/ref/models/fields/#model-field-types

Some optional field arguments are:

null: If True, Django will store empty values as NULL in the database. Default is False.

blank: If True, the field is allowed to be blank. Default is False.

choices: A sequence of 2-tuples to use as choices for this field. If this is given, the default form widget will be a select box instead of the standard text field and will limit choices to the choices given.

Choices example:

```
YEAR_IN_SCHOOL_CHOICES = [
    ("FR", "Freshman"),
    ("SO", "Sophomore"),
    ("JR", "Junior"),
    ("SR", "Senior"),
    ("GR", "Graduate"),
]
```

default: The default value for the field.

primary_key: If True, this field is the primary key for the model.

unique: If True, this field must be unique throughout the table.

Automatic primary key fields

By default, Django gives each model an auto-incrementing primary key with the type specified per app in AppConfig.default_auto_field or globally in the DEFAULT_AUTO_FIELD setting.

Example:
id = models.BigAutoField(primary_key=True)

Relationships

Many-to-one relationship: To define a many-to-one relationship, use django.db.models.ForeignKey. You use it just like any other Field type: by including it as a class attribute of your model.

Example:

theater = models.ForeignKey(Theater, on_delete=models.PROTECT) # In table screens

Many-to-many relationships: Use ManyToManyField. You use it just like any other Field type: by including it as a class attribute of your model.

Example:

actors = models.ManyToManyField(Actor) # In table movies

I have created some models already for the movies project. The models may change with the time as we will develop the project, and that is totally fine. I applied the models to the database.

Do you remember the commands to create migration files and apply those changes to your DB?

Please read https://docs.djangoproject.com/en/5.0/topics/db/models/ to find out more about Django models and create the models for your application.

*** Day 03 ***

Profile Table:

I am going to call "viewers" to the ones who will get tickets for my movie_theater. I have to link a "viewer" to a profile in the database. So I am going to create a ***Profile*** table. This table extends the ***User*** django table. It is better to not touch the User django table, you never know what the django developer may do in the future, so it is better to add a table and create a relationship OneToOne with a table that we are going to call ***Profile***, and this is the one we will use to link with a table for "viewer".

Create a core app:

python manage.py startapp core

In core/models.py, you must create the Profile table:

```
from django.db import models
from django.contrib.auth.models import User

from viewers.models import Viewer

# Create your models here.

class Profile(models.Model):
     user = models.OneToOneField(User, on_delete=models.CASCADE,
                                 primary_key=True)
     viewer = models.OneToOneField(Viewer, on_delete=models.CASCADE,
                                 blank=True, null=True)
     permissions_updated_at = models.DateTimeField()
     guid_password_reset = models.UUIDField(unique=True, null=True,
                                            blank=True)
     guid_password_expiry = models.DateTimeField(null=True,
                                                 blank=True)
```

In core, create templates/base.html (see the code provided)

Viewers App:

Create the viewers app:

python manage.py startapp viewers

In viewers/models.py, you must create the Viewer table:

```
from django.db import models


# Create your models here.

class Viewer(models.Model):
      name = models.CharField(max_length=200, unique=True)
```

DB Diagram:

You can generate a database diagram from Django. Please follow these directions to generate it. If you did not install the package pygraphviz because of some issues, you may generate it using our server lisa to host your project (documentation about it is provided in the tutorial about hosting your project on lisa).

The packages django-extensions and graphviz are needed to generate the database diagram directly from what you're writing in the models. See:
https://django-extensions.readthedocs.io/en/latest/graph_models.html

There are different options to do this, but the default one is to insert the following code in the settings.py:

```
GRAPH_MODELS = {
 'all_applications': True,
 'group_models': True,
}
```

And then you do execute this command in your terminal:

python manage.py graph_models -o database.png

And done you have your db diagrams saved as an image file: database.png.

Everytime you update your models, you can recreate the diagram to have an updated version.

Inserting data

ListView:

We are going to use some classes already built by django to insert, edit, and list data.

In the movie app, I am creating a view for actor and importing the Actor and Movie models:

```
from django.shortcuts import render
from django.views.generic import ListView
from django.views.generic.detail import DetailView
from django.views.generic.edit import CreateView, UpdateView
from django.urls import reverse_lazy
from django.contrib import messages

from .models import Actor, Movie

# Create your views here.

class ActorListView(ListView):
   model = Actor
```

For ListView, the documentation is there:
https://docs.djangoproject.com/en/4.2/ref/class-based-views/generic-display/#listview

You can customize multiple methods, but we are not going to do it in this class.

Create the template actor_list (templates/movies/actor_list.html). In the template, you can use the variable object_list to get the data.

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

{% block content %}

     {% for actor in object_list %}
          {{ actor.name }}<br/>
     {% endfor %}
{% endblock content %}
```

Create: urls for the movie app, urls in the project, load the movie urls. In settings.py do not forget to add core, movies and viewers to the INSTALLED_APPS list. (see the code I provided, I am no longer posting code here about the things you should know already).

If you go to http://localhost:8000/movies/actors/, there is nothing for the moment.

CreateView:

Let set class ActorCreateView(CreateView):

```
class ActorCreateView(CreateView):
    model = Actor
    fields = ['name']

    def form_valid(self, form):
        response = super().form_valid(form)
        messages.add_message(
            self.request, messages.SUCCESS,
            'Actor "{actor_name}" has been created'.format(
                actor_name=self.object.name))
        return response
```

Let's create the template movies/actor_form.html (this is the default template path to CreateView with model Actor).

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

{% block content %}
<form method="post">{% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Save">
</form>
{% endblock content %}
```

Do not forget to add it in the urls.

Go to the page, click on the button submit, and then Django will complain:

During handling of the above exception ('Actor' object has no attribute 'get_absolute_url')

ImproperlyConfigured at /movies/actors/new

No URL to redirect to. Either provide a url or define a get_absolute_url method on the Model.

On the documentation: https://docs.djangoproject.com/en/4.2/ref/class-based-views/generic-editing/#createview, there is nothing that talks about a url or get_absolute_url.

If you click on **ModelFormMixin**, you will see there is information about success_url or the method get_success_url(). The class **ModelFormMixin** will forward the success action to an url, and you need to give this information to ActorCreateView. About the other option, get_absolute_url, we will use it later for example for the model Movie.

I know that sometimes you do not know you could have some option available. There is a lot of Django documentation, and I do not know if they wrote everything there, but I know they tried to put all the information on it. Also, you do not know where to look because the documentation is huge. Well, better to have too much documentation than none, I can tell you. At some point, you have to be curious, and read and learn from tutorials.

Here in this case, you could search about it because Django was expecting information that we did not provide, so you could have been able to search about it.

Here you can read about ModelForm
https://docs.djangoproject.com/en/4.2/topics/class-based-views/generic-editing/#model-forms
and you will see that it explains about this success_url and the get_absolute_url()

Update your CreateView code to have the function get_success_url:

```
class ActorCreateView(CreateView):
      model = Actor
      fields = ['name']

      def form_valid(self, form):
            response = super().form_valid(form)
            messages.add_message(
                  self.request, messages.SUCCESS,
                  'Actor "{actor_name}" has been created'.format(
                        actor_name=self.object.name))
            return response

      def get_success_url(self):
            return reverse_lazy("movies:actor_detail",
                                args=[self.object.id])
      # you can also use it this way with kwargs, just to let you
know
      # but here we have only one argument, so no mistake can be done
      #return reverse_lazy("movies:actor_detail",
      #                    kwargs={'pk':self.object.id})
```

Let's create the detail view before defining the sucess_url: class ActorDetailView(DetailView)

```
class ActorDetailView(DetailView):
    model = Actor
```

Set up the template movies/actor_detail.html:

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

{% block content %}
     <a href="{% url 'movies:actor_list' %}">Actors
     list</a><br/><br/>
     <h1>{{ object.name }}</h1>
{% endblock content %}
```

Set up the urls.

Go to http://localhost:8000/movies/actors/1
And you will see the detail page for the actor with id 1.

We can change the list template to add the href of the detail url.
We can also add an href to the create webpage.

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

{% block content %}
     <a href="{% url 'movies:actor_create' %}">New
     actor</a><br/><br/>
     {% for actor in object_list %}
         <a href="{% url 'movies:actor_detail' actor.id %}">
            {{ actor.name }}
         </a><br/>
     {% endfor %}
{% endblock content %}
```

*** Day 04 ***

UpdateView:

This view is similar to the CreateView.

```
class ActorUpdateView(UpdateView):
     model = Actor
     fields = ['name']

     def form_valid(self, form):
          response = super().form_valid(form)
          messages.add_message(
                    self.request, messages.SUCCESS,
                    'Actor "{actor_name}" has been updated'.format(
                         actor_name = self.object.name))
          return response

     # the following method is to avoid the success_url error
     def get_success_url(self):
          return reverse_lazy("movies:actor_detail",
                              args=[self.object.id])
          # you can also have the return statement with kwargs
          # but here we have only one argument,
          # so no mistakes can be done
          #return reverse_lazy("movies:actor_detail",
          #                    kwargs={'pk':self.object.id})
```

Add the url:

```
path("actors/update/<int:pk>", views.ActorUpdateView.as_view(),
     name="actor_update")
```

You may add a link to update a record in the detail template:

```
<a href="{% url 'movies:actor_update' object.id %}">Update
Actor</a><br/>
```

Depending on the purpose of your web page, sometimes you may need to sort the list of records by a specific field, for example by name. For this, you must set it up on the meta class of the model:

```
class Actor(models.Model):
      name = models.CharField(max_length=200, unique=True)

      def __str__(self):
            return self.name

      class Meta:
            ordering = ['name']
```

Notice, I also included a __str__ function. This function converts an object to string. In this example, I need to have a list of actors in a dropdown menu when I create movies. So, I can see the names instead of Object(1).

DeleteView:

Make sure you have imported DeleteView:

```
from django.views.generic.edit import CreateView, UpdateView, DeleteView
```

The DeleteView looks like this:

```
class ActorDeleteView(DeleteView):
      model = Actor
      success_url = reverse_lazy("movies:actor_list")

      def form_valid(self, form):
            response = super().form_valid(form)
            messages.add_message(
                  self.request, messages.SUCCESS,
                  'Actor "{actor_name}" has been deleted'.format(
                        actor_name=self.object.name))
            return response
```

Create a actor_confirm_delete.html template:

```
{% block content %}
<form method="post">{% csrf_token %}
     <p>Are you sure you want to delete the actor "{{ object.name
}}"?</p>
     {{ form.as_p }}
     <input type="submit" value="Confirm">
</form>
{% endblock content %}
```

Add the url:

```
path("actors/delete/<int:pk>", views.ActorDeleteView.as_view(),
     name="actor_delete")
```

You may add a link to delete a record in the detail template:

```
<a href="{% url 'movies:actor_delete' object.id %}">Delete
Actor</a><br/>
```

See the code provided: Actors and Movies views are done.

*** Day 05 ***

Jupyter Notebook:

Jupyter Notebooks are useful to try Python code. You can write code, test it, modify the code and easily run it again.

Add jupyter to the requirements dev file.

Create a folder **notebooks** in your project directory (same level as manage.py). You may create your notebooks inside that folder. Notebooks have extension .ipynb. Example: api.ipynb (ipython notebook is the ancestor of jupyter notebook).

If using VSCodium, at some point it will ask you to install a jupyter extension, please do it, and then select the correct interpreter to run the kernel.

In the first cell, write this:

```
import os
os.chdir("..")
```

Run that code only once when you restart the kernel. (no more than once, otherwise the directory to load the module is going to a parent directory each time, and it will not be able to load the modules you need).

In the second cell, you can write the following code to load the Django environment:

```
import django
# In case that we need it later
# from django.conf import settings
os.environ.setdefault('DJANGO_SETTINGS_MODULE',
'movie_theater.settings')
# This is for async, in case we will see it later (maybe)
# os.environ["DJANGO_ALLOW_ASYNC_UNSAFE"] = "true"
django.setup()
```

Save data from an API:

Let's use 2 APIs to have some information about the movie on this page:
https://www.imdb.com/chart/top/

First API: https://rapidapi.com/apidojo/api/imdb8
Here, use the title/get-top-rated-movies to get the top 250 list. Be careful on using this api, you only have 500 requests per month.

Once you have the list of ids (the one that begins with tt), then we can use this api: https://www.omdbapi.com/ to retrieve the information about the movies: name, director, release date, runtime and the actors. (with this api, you have 1000 requests per day, in case you can save the raw data you ask in a file so you can play with it without being worrying about the queries limit). Create an account to have a key: Click on **"API Key"** and add your info for the account. You will receive an email to activate your account and use the api (example: **http://www.omdbapi.com/?i=tt0944947&plot=full&apikey=YOUR_KEY_HERE)**

I will let you figure out how to save this data on the database.
To start, search in the Django documentation how to save one object (record).
Once you know how to save an object, I would like you to research and use the functions: bulk_create and bulk_update to insert or update a lot of data at the same time (use a batch_size of 10 with our small database server, otherwise in a bigger server you could use a batch_size of thousands). You can use **pandas** if you would like. Do not forget you need to insert first the actors and then the movies, you will need the actor ids.