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)
      # you may override that with the following line
      # id = models.SmallIntegerField(auto_increment=True,
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.