BIT2008A/ITEC2000A: Multimedia Data Management
Fall 2022

Lab 9

Django Admin

As you may have noticed, a Django project initially has an app called **Admin**. There is a separate file named *admin.py* for us to configure and its URLs are managed under the *'admin/'* path. You can use it to quickly add, delete, or edit any database model from a web interface only by registering those models. But, you can customize Django admin to take its capabilities to the next level.
This tutorial aims to:

- Add attribute columns in the model object list
- Link between model objects
- Add filters to the model object list
- Make model object lists searchable
- Modify the object edit forms
- Override Django admin templates

Instructions

Please follow these step-by-step instructions:

i) Start with a sample Django project from Week 6 or create a new one. The command to start a new project is:

```
django-admin startproject <project name>
```

ii) Now, if you are using Docker, run your containers via the command below:

```
docker compose up
```

Then, add a new Django app to your project and create a superuser like below:

```
python manage.py startapp core
python manage.py migrate
python manage.py createsuperuser
```

Up to this point, you can log into the admin panel of your project.

iii) After adding a Django app, we should list it's name in *settings.py* like below:

```
INSTALLED_APPS = [
'django.contrib.admin',
'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',
'core.apps.CoreConfig' # new line
]
```

iv) After we have configured our app, we usually create our model classes. We want to create three models for **Person**, **Course**, and **Grade**. Your *models.py* should look like this:

```
from django.core.validators import MinValueValidator, MaxValueValidator
from django.db import models

class Person(models.Model):
    last_name = models.TextField()
    first_name = models.TextField()
    courses = models.ManyToManyField("Course", blank=True, related_name="
    students")

    class Meta:
        verbose_name_plural = "People"

class Course(models.Model):
    name = models.TextField()
    year = models.IntegerField()

    class Meta:
        unique_together = ("name", "year", )

class Grade(models.Model):
    person = models.ForeignKey(Person, on_delete=models.CASCADE)
    grade = models.PositiveSmallIntegerField(
        validators=[MinValueValidator(0), MaxValueValidator(100)])
    course = models.ForeignKey(Course, on_delete=models.CASCADE)
```

There are a couple of new options used in our models implementation:

- We have a Meta class to add metadata to our models. *verbose_name* takes care of the human-readable name for the object when singular. by default Django uses **class name** as the *verbose_name* for the model **ClassName**. For the plural form, it adds "s" to the name that does not work for **Person**. As a result, we assigned **People** to *verbose_name_plural*.
- We have the option *unique_together* for a set of field names that, taken together, must be unique.
- We have a list of default validators in Django to maintain some criteria to the values that we want to insert into our database. In this project, we set a lower bound and an upper bound to the grades using **MinValueValidator** and **MaxValueValidator**.

v) After the declaration of our models, we have to register them inside *admins.py* to include them in the admin panel. So far, we use the most basic form to register our apps.

```
from django.contrib import admin
from . import models

admin.site.register(models.Course)
admin.site.register(models.Person)
admin.site.register(models.Grade)
```

vi) Don't forget to reflect new changes to the database:

```
python manage.py makemigrations
python manage.py migrate
```

vii) We can reach to the admin panel via *http://localhost:8000/admin*. Right now if you try to add new instances to *People* or any classes, they will not be displayed appropriately. To change that, we have to override the string representation of the classes. **Add** these lines to your models.

```
class Person(models.Model):
    ...
    def __str__(self):
        return f"{self.last_name}, {self.first_name}"

class Course(models.Model):
    ...

    def __str__(self):
        return f"{self.name}, {self.year}"

class Grade(models.Model):
    ...

    def __str__(self):
        return f"{self.grade}, {self.person}, {self.course}"
```

viii) Implementing __str__() is a quick way to change the representation of a Person object from a meaningless string to understandable data. Since this representation will also show up in drop-downs and multi-selects, you definitely want to make it as easy to understand as possible.

You can customize change list pages in far more ways than just modifying an object's string representation. The *list_display* attribute of an *admin.ModelAdmin* object specifies what columns are shown in the change list. This value is a tuple of attributes of the object being modeled. For example, in *core/admin.py*, modify **PersonAdmin** as follows: (replace it with our simple registration of **Person**)

```
@admin.register(models.Person)
class PersonAdmin(admin.ModelAdmin):
    list_display = ("last_name", "first_name")
```

ix) The *list_display* tuple can reference any attribute of the object being listed. It can also reference a method in the *admin.ModelAdmin* itself. Modify **PersonAdmin** again: **Person**)

```
from django.db.models import Avg


@admin.register(models.Person)
class PersonAdmin(admin.ModelAdmin):
    list_display = ("last_name", "first_name", "show_average")

    def show_average(self, obj):
        result = models.Grade.objects.filter(person=obj).aggregate(Avg("grade
    "))
        return result["grade__avg"]

    show_average.short_description = "Average Grade"
```

In the code above, you add a column to the admin that displays each student's grade average. *show_average()* is called once for each object displayed in the list. The parameter *obj* is the object of the row that is displayed. In this case, you use it to query the corresponding Grade objects for the student, with the response averaged over *Grade.grade* and finally, *short_description* changes the displayed name of your new column.

x) Now, let us enhance the representation of **Course** class. It's common for objects to reference other objects through the use of foreign keys. You can point *list_display* at a method that returns an HTML link. Now, replace the initial **Course** registry with the **CourseAdmin** snippet as follows:

```
from django.utils.html import format_html
```

```
from django.urls import reverse
from django.utils.http import urlencode

@admin.register(models.Course)
class CourseAdmin(admin.ModelAdmin):
    list_display = ("name", "year", "view_students_link")

    def view_students_link(self, obj):
        count = obj.students.count() # related_name
        url = (
            reverse("admin:core_person_changelist")
            + "?"
            + urlencode({"courses__id": f"{obj.id}"})
        )
        return format_html(f'<a href="{url}">{count} Students</a>')

    view_students_link.short_description = "Students"
```

In the code above, we use *reverse()* to look up a URL in the Django admin. There is a convention to find any admin page but for now, we managed to create the URL address to redirect us the list of **Person** objects, filtered by desired course ID. We have to use *format_html* to use HTML links outside of our templates.

xi) Afterward, let us configure the Course admin page by adding a filter to the list screen. Add the line below to have *list_filter*:

```
@admin.register(models.Course)
class CourseAdmin(admin.ModelAdmin):
    list_display = ("name", "year", "view_students_link")
    list_filter = ("year", )
    ...
```

Now, you can only display the courses of the year that you are looking for.

xii) For the **Person** class, we can assign a *search box* to filter its objects. Add the line below to implement the following: You can customize more than just the change list page. The screens used to add or change an object are based on a ModelForm. Django automatically generates the form based on the model being edited.

You can control which fields are included, as well as their order, by editing the fields option. Modify your PersonAdmin object, adding a field attribute:

xiii) You can customize more than just the change list page. The screens used to add or change an object are based on **ModelForm**. Django automatically generates the form based on the model being edited. You can control which fields are included, as well as their order, by editing the **fields** option. Modify your PersonAdmin object, adding a fields attribute to bring *first name* before *last name*:

```
@admin.register(models.Person)
class PersonAdmin(admin.ModelAdmin):
    fields = ("first_name", "last_name", "courses")
    ...
```

ModelAdmin.get_form() is responsible for creating **ModelForm** for your object. You can override this method to change the form. Add the following method to PersonAdmin to add labels to first-name text input:

```
@admin.register(models.Person)
class PersonAdmin(admin.ModelAdmin):
    ...
    def get_form(self, request, obj=None, **kwargs):
        form = super().get_form(request, obj, **kwargs)
```

```
        form.base_fields["first_name"].label = "First Name (Humans only!):"
        return form
    ...
```

xiv) In some scenarios, changing the label might not be sufficient. On the other hand, we might not have access to, or want to change the model class to perform certain evaluations. Then we can use the form attribute to register a custom form with its own evaluations. Make the following additions and changes to *admin.py*:

```
from django import forms

class PersonAdminForm(forms.ModelForm):
    class Meta:
        model = models.Person
        fields = "__all__"

    def clean_first_name(self):
        if self.cleaned_data["first_name"] == "Spike":
            raise forms.ValidationError("No Vampires")

        return self.cleaned_data["first_name"]


@admin.register(Person)
class PersonAdmin(admin.ModelAdmin):
    form = PersonAdminForm
    ...
```

xv) You can fully customize the admin templates by changing or extending its default templates used to render pages. You can find all the templates used in the admin by looking inside the Django package. Let us customize the logout page. The relative path leading to the file must be the same as the overridden path. The file you're interested in is **registration/logged_out.html**. Start by creating the directory in the project's root directory and add it to *settings.py*:

```
TEMPLATES = [
    {
    ...
        # Add the templates directory to the DIR option:
        "DIRS": [os.path.join(BASE_DIR, "templates"), ],
    ...
    }
]
```

The template engine searches directories in the DIR option before the application directories, so anything with the same name as an admin template will be loaded instead. To see this in action, copy the **logged_out.html** file into your **templates/registration** directory and then modify it to add these two paragraphs:

```
{% extends "admin/base_site.html" %}
{% load i18n %}

{% block breadcrumbs %}<div class="breadcrumbs"><a href="{% url 'admin:index'
    %}">{% trans 'Home' %}</a></div>{% endblock %}

{% block content %}

<p>You are now leaving Sunnydale</p>

```

```
<p><a href="{% url 'admin:index' %}">{% trans 'Log in again' %}</a></p>

{% endblock %}
```

References

- The instructions to this session was brought from this tutorial.
- If you wish to modify the admin panel of the Poll project (Lab 6), you can follow the instructions of this tutorial.