jueves, 24 de noviembre de 2022

Clase 2: Crear aplicación

Objetivo: Crear una aplicación.

Nota importante: Desde este tutorial, usaré en los comandos a menudo "`docker compose exec <servicio>`". Esta porción es para ejecutar comandos internamente en la máquina virtual (contenedor) del servicio que indique. Es probable que en ocasiones no ponga explícitamente el comando, dado que en muy pocas ocasiones necesitaremos ejecutar comandos fuera de nuestro contenedor, ustedes deben notar si el comando debe ejecutarse dentro del contenedor o no ya que será bastante obvio. Otra forma en que aparecerá este comando es en esta: `"docker compose exec <servicio> bash"` lo que es similar, con diferencia que este comando nos dejará dentro del contenedor en una terminal bash.

---

1. Migraciones pendientes

Si revisan los logs del servicio web pueden ver lo siguiente:

```
> docker compose logs -f web
taller  | Watching for file changes with StatReloader
taller  | Performing system checks...
taller  |
taller  | System check identified no issues (0 silenced).
taller  |
taller  | You have 18 unapplied migration(s). Your project may
not work properly until you apply the migrations for app(s):
admin, auth, contenttypes, sessions.
taller  | Run 'python manage.py migrate' to apply them.
taller  | November 30, 2022 - 13:04:04
taller  | Django version 3.2.16, using settings
'taller.settings'
taller  | Starting development server at http://0.0.0.0:8000/
taller  | Quit the server with CONTROL-C.
```

Ahí dice que tenemos 18 migraciones sin aplicar. ¿Qué es una migración? Un archivo con código Python con una "receta" para ejecutar acciones en la base de datos.

Si hacemos:

```
 > docker compose exec web python manage.py showmigrations
```

Podremos ver una lista de las migraciones detectadas. Las con el símbolo **[*]** son las migraciones que ya han sido aplicadas a la base de datos. Como vemos, no se ha aplicado ninguna:

```
admin
 [ ] 0001_initial
 [ ] 0002_logentry_remove_auto_add
 [ ] 0003_logentry_add_action_flag_choices
auth
 [ ] 0001_initial
 [ ] 0002_alter_permission_name_max_length
 [ ] 0003_alter_user_email_max_length
 [ ] 0004_alter_user_username_opts
 [ ] 0005_alter_user_last_login_null
 [ ] 0006_require_contenttypes_0002
 [ ] 0007_alter_validators_add_error_messages
 [ ] 0008_alter_user_username_max_length
 [ ] 0009_alter_user_last_name_max_length
 [ ] 0010_alter_group_name_max_length
 [ ] 0011_update_proxy_permissions
 [ ] 0012_alter_user_first_name_max_length
contenttypes
 [ ] 0001_initial
 [ ] 0002_remove_content_type_name
sessions
 [ ] 0001_initial
```

Las entradas “admin”, “auth”, “content types” y “sessions” son aplicaciones que Django ya trae pre-instaladas. Estas estarán creando tablas en la base de datos y haciendo cambios, los que están especificados en esos archivos python.

Para que estas migraciones se ejecuten necesitamos ejecutar este comando:

```
docker compose exec web python manage.py migrate
```

---

1. Difinir qué hará nuestra aplicación.

Vamos a crear nuestra primera aplicación. Una aplicación de Django es un conjunto de archivos de Python que van a tener funcionalidades con un propósito en común.

El proyecto: En nuestro caso, nuestro proyecto se trata de un sistema que permita la administración del aforo para uno o varios edificios. La idea es que las personas puedan registrarse a un evento en concreto que se realizará un edificio determinado. Una aplicación de Django aquí podría encapsular todos los archivos relacionados con la captura y administración de las inscripciones.

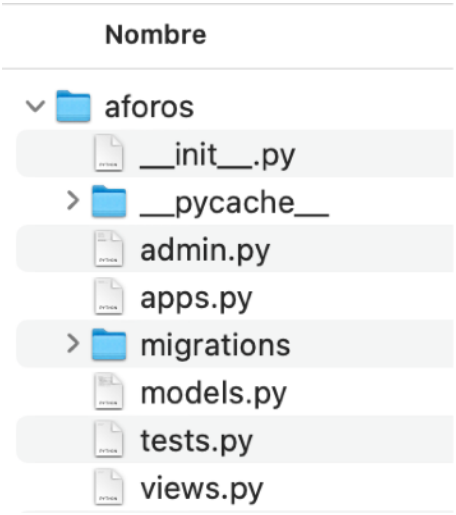
---

2. Crear la aplicación

Para crear la aplicación necesitaremos abrir una terminar en la carpeta del proyecto y escribir:

```
# Entrar al contenedor
> docker compose exec web bash
> python manage.py startapp aforos
```

Esto debería haber creado la siguiente carpeta con estos archivos:



Para que Django tome en cuenta nuestra aplicación, necesitamos agregarla a los ajustes del proyecto. Así que vamos a editar en el archivo de configuración **(/taller/settings.py)** la variable INSTALLED_APPS y vamos a agregar la aplicación “aforos” como aparece aquí:

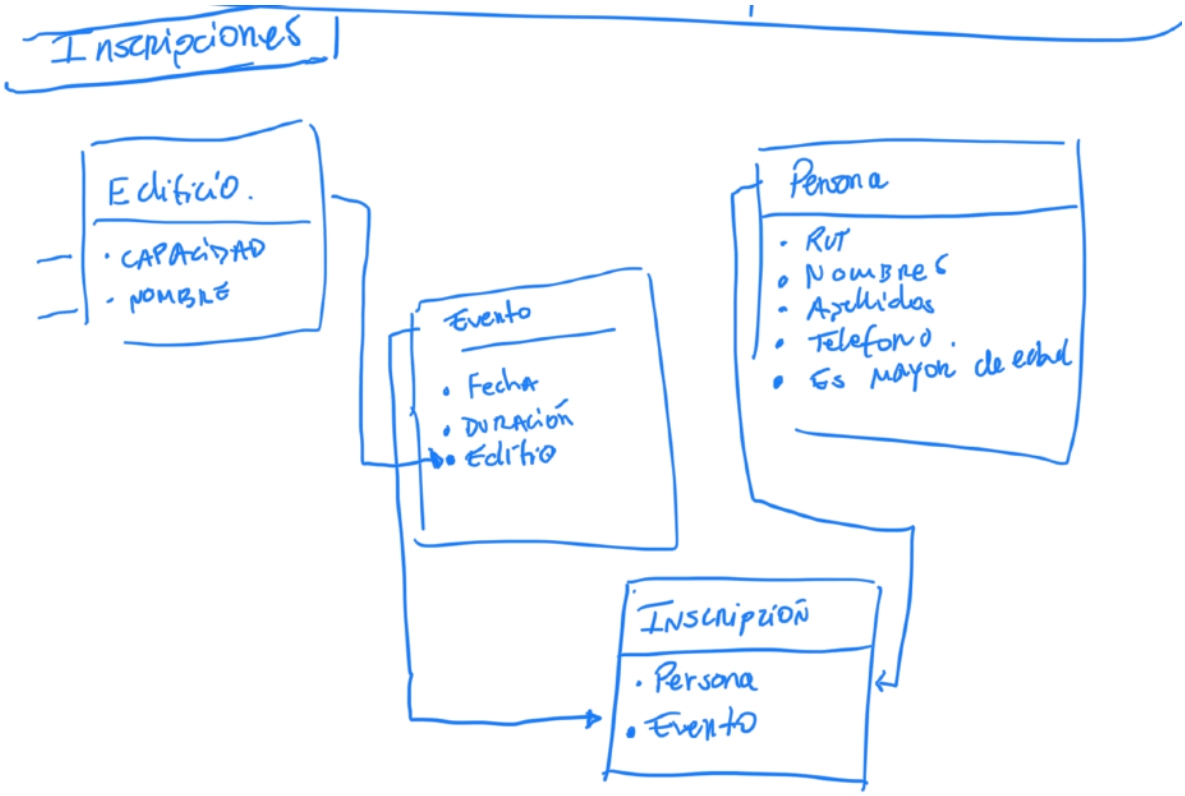
```
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'aforos'
]
```

Con esto nuestra aplicación ya será considerada para la base de datos y las urls del proyecto.

---

3. Agregar primeros modelos

Para la aplicación queremos agregar tablas de base de datos, así que tomando en cuenta los objetivos de la aplicación diseñamos la siguiente estructura (muy simple) de base de datos relacional, les comparto el esbozo que hicimos en clases:



Claramente, esto no es un diseño exacto de nuestra base de datos, sólo incluimos los datos que deseamos guardar. Si quisiéramos hacer un diseño exacto, deberíamos incluir símbolos más precisos y los campos identificadores de las tablas. Pero Django puede aceptar este tipo de definiciones simples, y si no indicamos cuál es nuestro *Primary Key* (concepto de base de datos) se agregará a cada tabla un campo llamado “ID” de forma automática.

Para que Django sepa qué tablas debe crear y qué hacer con la base de datos necesitamos editar el archivo **/aforos/models.py**, este contendrá las definiciones de nuestras tablas.

aforos > models.py > ...

```python
from django.db import models


class Edificio(models.Model):
    capacidad = models.PositiveIntegerField(help_text="Capacidad de aforo del edificio")
    nombre = models.CharField(max_length=200)


class Persona(models.Model):
    rut = models.CharField(max_length=9,
        help_text="Sin puntos ni guiones y con dígito verificador.",
        unique=True)
    nombres = models.CharField(max_length=100)
    apellidos = models.CharField(max_length=100)
    telefono = models.PositiveIntegerField(help_text="Sin código de país y con 9")
    es_menor = models.BooleanField(help_text="Indica si es un menor de edad")


class Evento(models.Model):
    fecha = models.DateTimeField()
    duracion = models.PositiveIntegerField(help_text="Cantidad de horas de duración del evento.")
    edificio = models.ForeignKey("Edificio", on_delete=models.CASCADE)


class Inscripcion(models.Model):
    persona = models.ForeignKey("Persona", on_delete=models.CASCADE)
    evento = models.ForeignKey("Evento", on_delete=models.CASCADE)
```

Detalles importantes. Aquí se usa *Programación Orientada a Objetos,* por lo que las tablas se definen como clases y los campos serán atributos de cada clase. En cada atributo, guardamos instancias de otras clases que Django ya tiene preparadas para nosotros (los que se llaman algo así: *models.…Field(…)*), eso va a definir qué tipo de datos será cada columna, por eso hay Bool, Char, PositiveInteger, que serán los tipos de datos que se le asignará a las columnas. También están los ForeignKeys, lo que permite hacer referencias a otras tablas que estamos creando.

Una vez con esto escrito, podremos crear las migraciones y ejecutarlas. Recordemos: las migraciones son recetas escritas en código Python que le indicarán a la base de datos qué tablas y columnas crear o modificar para tener la base de datos que queremos.

Para esto, ejecutamos dentro del contenedor:

```
> python manage.py makemigrations
```

Lo cual creará las migraciones, las podrán ver en la carpeta **/aforos/migrations/**

```
# Y luego para ejecutarlas:
> python manage.py migrate
```

Con esto, nuestras tablas estarán creadas y estaremos listos para guardar datos en ellas.

4. Sitio de administración

a. Para finalizar la clase crearemos una forma de interactuar con nuestras tablas. Django incluye herramientas para hacerlo muy rápido con un sitio web de administración por defecto. Para utilizarlo editaremos el archivo **/aforos/admin.py** como sigue a continuación:

```
aforos > admin.py > ...
1   from django.contrib import admin
2   from aforos.models import *
3
4   class EdificioAdmin(admin.ModelAdmin):
5       pass
6
7
8   class EventoAdmin(admin.ModelAdmin):
9       pass
10
11
12  class PersonaAdmin(admin.ModelAdmin):
13      pass
14
15
16  class InscripcionAdmin(admin.ModelAdmin):
17      pass
18
19  # Asociamos las clases de arriba, con los modelos que importamos de models.py
20  admin.site.register(Edificio, EdificioAdmin)
21  admin.site.register(Evento, EventoAdmin)
22  admin.site.register(Persona, PersonaAdmin)
23  admin.site.register(Inscripcion, InscripcionAdmin)
24
```

b. Aquí estamos importando desde el archivo **/aforos/models.py** todos los modelos (tablas) que definimos en el paso anterior (línea 2). De las líneas 4-17 definimos clases heredando una clase padre que importamos desde las bibliotecas de Django (la clase **admin.ModelAdmin**). Esto nos permitirá tener una clase para cada tabla con todas las funciones de la clase padre **ModelAdmin**.

La clase de la que heredamos **ModelAdmin** incluye todas las definiciones y funciones para crear a partir del modelo/tabla indicado una página de administración en el Sitio de Administración de Django. Esto quiere decir que a través de estas clases, automáticamente se crearán páginas para: listar, leer, crear, editar y eliminar objetos en nuestras tablas.

c. Lo único que nos resta hacer indicarle al sitio de administración que utilice estas clases que creamos para administrar sus correspondientes modelos/tablas. (Líneas 20-23). Por eso asociamos **Edificio con EdificioAdmin**, por ejemplo.

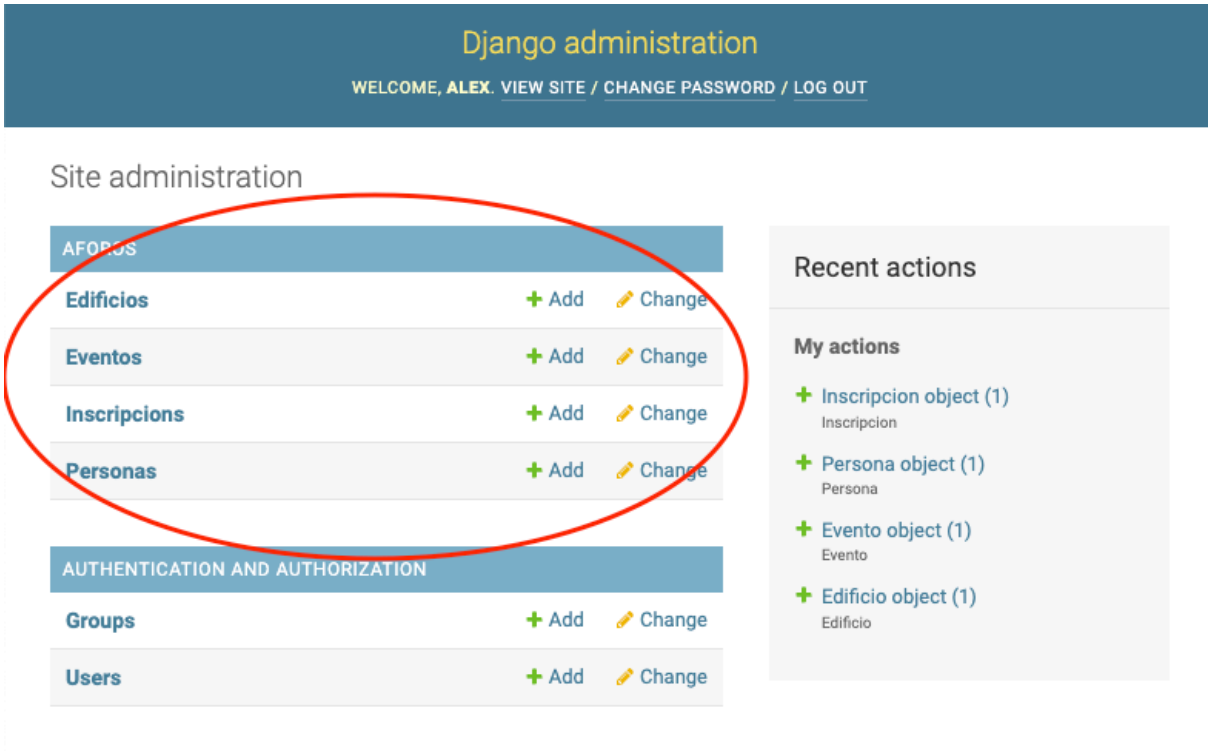
d. Al terminar, podemos acceder al sitio de administración yendo a este enlace en el navegador: **localhost:8000/admin/**

e. Ahora requerirá que iniciemos sesión. Ya que no tenemos usuario administrador aún, vamos a crear uno escribiendo este comando dentro del contenedor:

```
> python manage.py createsuperuser
```

f. Este nos asistirá para crear uno, en mi caso hice uno con nombre: **admin y password: admin.**

g. Una vez tenga un usuario, vuelvo al sitio de administración y podré encontrar esta sección listada:



Ustedes pueden explorar las posibilidades que les da el sitio de administración. Hasta aquí llega esta clase.