**CLASES DE DJANGO EN CODERHOUSE**

**CLASE 1: Clase 17 - Django - Portfolio Parte I**

**Entorno virtual.**

1. Creamos la carpeta donde trabajaremos en nuestro proyecto de Django.
2. Abrimos VsCode posicionándonos en dicha carpeta.
3. Si no tenemos instalado el gestor de entornos virtuales **pipenv**, abrimos la termina desde VsCode ctrl + j y lo instalamos:  
   pip install pipenv
4. Si queremos seleccionar una versión de Python específica para nuestro proyecto debemos tenerla instalada en la PC. Luego podemos crear el entorno con el siguiente comando:  
   pipenv --python 3.9.13
5. Usamos un mismo comando para crear o activar el entorno virtual:  
   pipenv shell  
   El entorno virtual queda indicado en la consola con un prefijo similar a esto:  
   (nombre\_entorno\_virtual) C:\path\_donde\_estamos\_ubicados  
   De no ser así, hay varias formas de solucionarlo, ver: **(1) Problemas**

**Instalaciones**

1. Creamos el archivo de **requirements.txt**, el cuál por el momento sólo va a tener la versión de Django que deseemos. También es buena práctica indicar a modo de comentario la versión de Python que usamos para el proyecto.
2. # Python 3.9.13
3. django==4.2
4. lo ejecutamos con pipenv:  
   pipenv install -r requirements.txt

**Django**

1. Creamos un proyecto de Django. Con el siguiente comando vamos a crear una carpeta en la posición en que nos encontramos y luego dentro de la misma se cargarán los archivos del proyecto:  
   django-admin startproject <nombre\_del\_proyecto>  
   Ej: django-admin startproject Clases\_Coder
2. Debido a que hemos creado una subcarpeta, debemos ingresar a la misma para poder interactuar con el archivo **manage.py** de Django:  
   cd <nombre\_del\_proyecto>  
   Ej: cd Clases\_Coder
3. Corremos las migraciones. Si bien no es obligatorio, éstas migraciones de Django van a crear las tablas necesarias para el Admin de Django, cuentas de usuario, etc. Asimismo con esta acción evitamos los mensajes de advertencia al iniciar el servidor:  
   python manage.py migrate
4. Opcional: Ya estamos en condiciones de correr el servidor y comprobar que el proyecto está correctamente creado. Con el siguiente comando, Django crea un servidor y lo corre bajo el puerto 8000:  
   * pyhton manage.py runserver
   * Nos dirigimos al navegador y colocamos localhost:8000 o bien mantemos presionado Ctrl y hacemos clic en el link que nos figura en la consola: [http://127.0.0.1:8000/http://127.0.0.1:8000/](http://127.0.0.1:8000/)

**Git**

**NOTA IMPORTANTE:** Todos los comandos que se hagan sobre Git, deben ejecutarse con la consola ubicada en la posición donde se encuentra la carpeta **.git** y no dentro del proyecto de Django, esto suele ser un error ya que los comandos que ejecutemos de Git va a tener dificultades con algunos comandos si generamos cambios en carpetas que no puede ver.

1. Creamos el repositorio ejecutando en la consola: git init
2. Creamos en la misma carpeta el archivo **.gitignore**. NOTA: Debería agregarse la base de datos para evitar que se suba como parte del repositorio, pero para el curso vamos a subirla para quién requiera descargar el repositorio ya cuente con datos agregados.
3. # Archivos de compilación de Python
4. \_\_pycache\_\_/
5. \*.pyc
6. # Archivos de pipenv. Sólo dejamos requirements.txt
7. Pipfile
8. Pipfile.lock
9. # Deberíamos agregar la base de datos con:
10. # db.sqlite3
11. Pasamos al **staging area** los archivos **.gitignore** y **requirements.txt**:  
    git add .
12. Hacemos el primer commit:  
    git commit -m "Primer commit"
13. Linkeamos nuestro repositorio de Git con el que hemos creado en Github: git remote add origin https://github.com/<mi\_cuenta\_github>/<nombre\_repositorio>.git
14. Si en local tenemos la rama **master**, vamos a cambiarle el nombre por **main**: git branch -M main
15. Subimos los cambios que hicimos en local al repositorio de Github: git push -u origin main
16. Creamos una nueva rama a partir de ésta instancia del **main** y nos movemos a la nueva rama, en nuestro ejemplo se va a llamar **clase\_17-Django\_Portfolio\_Parte\_1**:  
    git checkout -b clase\_17-Django\_Portfolio\_Parte\_1

**Contenido de la clase**

**Resumen:** Vamos a crear nuestras primeras interacciones desde el front con nuestro back. Esto permitirá que modificando la url del navegador, se pueda dirigir a una página simple pero con el contenido que nosotros queremos compartir.

1. Primero creamos la función de Python que se encargará de devolver el código html al front. Para ello, debemos ingresar a la carpeta de configuración del proyecto y agregar un nuevo archivo llamado **views.py**. La carpeta de configuración del proyecto se encuentra dentro de la carpeta del propio proyecto con su mismo nombre, podemos distinguirla porque dicha carpeta contiene los archivos: \_\_init\_\_.py, asgi.py, settings.py, urls.py, wsgi.py
2. En el archivo que acabamos de crear **views.py**, agregamos una función que será la encargada de crear el archivo html y devolvérselo al front. Para el ejemplo vamos a crear la función llamada **saludo** y sólo devolverá un texto al front. Debe contener:
3. from django.http import HttpResponse
4. def saludo(request):
5. return HttpResponse("Hola mundo!, hola Coder!")
6. En la misma carpeta, tenemos el archivo **urls.py**, vamos a editarlo para que la url apunte a la función que acabamos de crear. Por lo tanto, vamos a importar la función (from .views import saludo) y luego agregamos el **path** dentro de la lista de path's llamada **urlpatterns** para definir la url que se debrerá ingresar en el navegador y que la misma nos redirija a dicha función (path('saludo/', saludo)). El archivo debe quedar de la siguiente manera:
7. from django.contrib import admin
8. from django.urls import path
9. from .views import saludo
10. urlpatterns = [
11. path('admin/', admin.site.urls),
12. path('saludo/', saludo),
13. ]
14. Podemos comprobar los resultados levantando el servidor ejecutando en la consola: python manage.py runserver, y luego agregando a la url dentro del navegador la nueva página: localhost:8000/saludo/
15. Pasaje de parámetros:  
    A continuación vamos a enviar datos procesados con Python a la vista, si bien en el ejemplo sólo vamos a mandar el momento del día en que se ejecuta la vista, el proceso puede representar cualquier dato extraído de métodos o funciones internas que luego compartiremos con el front.
16. En **urls.py** agregamos el path diaDeHoy/:
17. from django.contrib import admin
18. from django.urls import path
19. from Clases\_Coder.views import saludo, dia\_de\_hoy
20. urlpatterns = [
21. path('admin/', admin.site.urls),
22. path('saludo/', saludo),
23. path('diaDeHoy/', dia\_de\_hoy),
24. ]
25. En **views.py** agregamos el método dia\_de\_hoy():
26. from django.http import HttpResponse
27. *# Como vamos a usar datetime para obtener el momento, importamos la librería*
28. from datetime import datetime
29. def dia\_de\_hoy(request):
30. dia = datetime.now()
31. return HttpResponse(f"Hoy es día:<br>{dia}")
32. Ya podemos consultar la web y ver sus resultados!: [**http://127.0.0.1:8000/diaDeHoy/**](http://127.0.0.1:8000/diaDeHoy/)
33. Enviando valores usando el método GET:  
    Ahora seguiremos los mismos pasos, pero vamos a enviar información desde el front hacia nuestro back mediante el método GET, el cuál se ejecuta cuando modificamos la url, por lo tanto, en la url vamos a incorporar una variable. En nuestro ejemplo vamos a enviar un nombre y el back creará una página con dicha información. Cabe aclarar que el back puede utilizar ese dato para realizar diversas tareas, como buscar en la base de datos, realizar peticiones a otros servicios, etc.  
    Vamos a agregar la función dentro de **views.py**, y dentro de sus parámetros vamos a recibir la variable llamada **nombre**:
34. def muestra\_nombre(request, nombre):
35. return HttpResponse(f"Buenos días {nombre}, bienvenido a Coder")
36. En el archivo **urls.py** importamos la función y creamos el path dentro de la lista **urlpatterns** para que nos redirija a la función recién creada y además le indicamos que pueda recibir un parámetro extra bajo el nombre de variable **nombre**:
37. from django.contrib import admin
38. from django.urls import path
39. from .views import saludo, muestra\_nombre
40. urlpatterns = [
41. path('admin/', admin.site.urls),
42. path('saludo/', saludo),
43. path('muestra\_nombre/<nombre>/', muestra\_nombre),
44. ]
45. Ya podemos probarlo modificando la url del navegador: localhost:8000//muestra\_nombre/Coder  
    Debería aparecer la frase que creamos en la función del **views.py** con el nombre de "Coder", el cuál podemos modificarlo sin problemas desde la url del navegador.
46. Usando plantillas:  
    A continuación vamos a crear un archivo html y que sea dicho archivo el que se envíe al front.  
    Para esto, vamos a crear en la misma carpeta que venimos trabajando, una nueva carpeta con el nombre que deseen, para el ejemplo se llamará **plantillas**.
47. Dentro vamos a crear un archivo html, para el ejemplo se llamará **index.html**.
48. <!DOCTYPE html>
49. <html lang="en">
50. <head>
51. <meta charset="UTF-8">
52. <meta http-equiv="X-UA-Compatible" content="IE=edge">
53. <meta name="viewport" content="width=device-width, initial-scale=1.0">
54. <title>Document</title>
55. </head>
56. <body>
57. Hola, esta es nuestra primer plantilla!
58. </body>
59. </html>
60. Generamos una nueva vista, o sea, una nueva función en el archivo **views.py** que en vez de retornar texto como lo hicimos anteriormente, vamos a retornar el archivo html. Como el código se empieza a extender, sólo mostramos los agregados al archivo y no el contenido completo:
61. *# Agregamos al encabezado del archivo el import de Template y de Context*
62. from django.template import Template, Context
63. def probando\_template(request):
64. *# Abrimos el archivo html*
65. mi\_html = open('./Clases\_Coder/plantillas/index.html')
66. *# Creamos el template haciendo uso de la clase Template*
67. plantilla = Template(mi\_html.read())
68. *# Cerramos el archivo previamente abierto, ya que lo tenemos cargado en la variable plantilla*
69. mi\_html.close()
70. *# Creamos un contexto, más adelante vamos a aprender a usarlo, ahora lo necesitamos aunque sea vacío para que funcione*
71. mi\_contexto = Context()
72. *# Terminamos de construír el template renderizándolo con su contexto*
73. documento = plantilla.render(mi\_contexto)
74. return HttpResponse(documento)
75. Editamos nuestro archivo **urls.py** para crear una url que nos redirija a la función recién creada:
76. from django.contrib import admin
77. from django.urls import path
78. from .views import saludo, muestra\_nombre, probando\_template
79. urlpatterns = [
80. path('admin/', admin.site.urls),
81. path('saludo/', saludo),
82. path('muestra\_nombre/<nombre>/', muestra\_nombre),
83. path('probando\_template/', probando\_template),
84. ]
85. Ya podemos probar en nuestro navegador ingresando: http://localhost:8000/probando\_template/
86. Desde la consola, vamos a detener el servidor presionando: Ctrl + c

**Subimos los cambios a GitHub**

1. Subimos los cambios a nuestro repositorio de GitHub:
   * git status
   * git add .
   * git commit -m "Paso 1 del proyecto: 1er clase"
   * git push --set-upstream origin clase\_17-Django\_Portfolio\_Parte\_I
2. En Github realizamos un **Pull Request** (de ahora en más PR) y hacemos el merge a **main**.

**CLASES DE DJANGO EN CODERHOUSE**

**PRE CLASE 2: Clase 18 - Portfolio Parte II**

**Entorno virtual**

De ahora en más, vamos a interpretar que el entorno virtual es activado en cada clase.

**Git**

1. Nos vamos a mover a la rama main, actualizarla y crear una nueva rama para la nueva clase: git checkout main
2. Actualizamos con: git pull. Esto se descarga los cambios que hayan en la rama main. Recordemos que el PR realizado en la clase anterior agregó archivos y modificaciones en la rama main.
3. Creamos y nos movemos a la nueva rama: git checkout -b clase\_18-Portofio\_Parte\_II

**Django: Templates**

Resumen:  
Vamos a crear datos propios en el back para mostrarlos en el front haciendo uso de un diccionario como contexto. Este diccionario se procesa para ser renderizado junto con el template (archivo html) y enviado al front.  
Cabe aclarar que en el ejemplo sólo vamos a crear un diccionario con datos estáticos (o como decimos en la jerga, valores hardcodeados), pero los mismos pueden ser construídos a partir de cualquier acción que deseemos en nuestras funciones, como obtenerlos desde la base de datos, operaciones internas de nuestro desarrollo, etc.

**Agregamos diccionarios**

1. Agregamos a la vista (**views.py**) un diccionario con datos de nombre y apellido y luego lo agregamos al contexto. El resto del código permanece igual:
2. def probando\_template(request):
3. nombre = "Adrian"
4. apellido = "Holovaty"
5. diccionario = {"nombre": nombre, "apellido": apellido}
6. *# Abrimos el archivo html*
7. mi\_html = open('./Clases\_Coder/plantillas/index.html')
8. *# Creamos el template haciendo uso de la clase Template*
9. plantilla = Template(mi\_html.read())
10. *# Cerramos el archivo previamente abierto, ya que lo tenemos cargado en la variable plantilla*
11. mi\_html.close()
12. *# Creamos un contexto con los datos del diccionario*
13. mi\_contexto = Context(diccionario)
14. *# Terminamos de construír el template renderizándolo con su contexto*
15. documento = plantilla.render(mi\_contexto)
16. return HttpResponse(documento)
17. En el archivo html, vamos a hacer uso del contexto pero accedemos al mismo con una sintaxis especial y como si fueran variables. Es decir, si queremos acceder al contenido del apellido, sólo vamos a nombrar la key del diccionario que creamos dentro de 2 llaves:
18. <!DOCTYPE html>
19. <html lang="en">
20. <head>
21. <meta charset="UTF-8">
22. <meta http-equiv="X-UA-Compatible" content="IE=edge">
23. <meta name="viewport" content="width=device-width, initial-scale=1.0">
24. <title>Document</title>
25. </head>
26. <body>
27. <p style="color: red"> Hola {{nombre}}, estamos haciendo uso de variables por contexto</p>
28. <p style="color: green">Y podemos acceder al apellido "{{apellido}}" porque tenemos acceso a ambas variables.<br>
29. También aplicamos estilos pero luego lo haremos con CSS porque no es buena práctica hacerlo en el archivo html.</p>
30. </body>
31. </html>

**Enviamos una lista y recorremos con bucle**

1. Ahora vamos a enviar una lista de datos y mostrarla con un bucle. Agregamos al diccionario una key adicional con una lista, quedando el diccionario de la siguiente manera:  
   diccionario = {"nombre": nombre, "apellido": apellido, "notas": [4, 8, 9, 10, 7, 8]}
2. Así como para acceder a valores dentro de un template usamos dobles llaves, para ejecutar código debemos usar la sintaxis de llaves y símbolo de porcentaje. A continuación, creamos un bucle for en el template para mostrar la lista de notas:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Document</title>

</head>

<body>

<p style="color: red"> Hola {{nombre}}, estamos haciendo uso de variables por contexto</p>

<p style="color: green">Y podemos acceder al apellido "{{apellido}}" porque tenemos acceso a ambas variables.<br>

También aplicamos estilos pero luego lo haremos con CSS porque no es buena práctica hacerlo en el archivo html.</p>

<h2>Notas:</h2>

{% for n in notas %}

<p>Nota: {{n}}</p>

{% endfor %}

</body>

</html>

**Agregamos condicional**

1. Además podemos agregar un condicional if, ejemplo:

{% for n in notas %}

{% if n <= 4 %}

<p style="color: red">Reprobado: {{n}}</p>

{% else %}

<p style="color: green">Aprobado: {{n}}</p>

{% endif %}

{% endfor %}

**Agregamos loader (cargadores)**

1. Ahora vamos a simplificar el código para cargar plantillas, ya que ésto puede ser tedioso y laborioso. Primero debemos importar en el archivo **views.py** el método que nos permite hacer esto: from django.template import loader
2. Luego, vamos a crear una nueva vista en el mismo archivo así podemos comparar la diferencia con el anterior método.
3. def usando\_loader(request):
4. diccionario = {
5. "nombre": nombre,
6. "apellido": apellido,
7. "notas": [4, 8, 9, 10, 7, 8]
8. }
9. plantilla = loader.get\_template('index.html')
10. documento = plantilla.render(diccionario)
11. return HttpResponse(documento)
12. Debido a que la forma de indicar el archivo html es diferente para este caso, lo que nos falta hacer es editar la lista **TEMPLATES**, y dentro de su diccionario la key **DIRS** que también contiene una lista. Dicha lista se encuentra en el archivo **settings.py**. En ella, podemos indicar el path de todas las carpetas que tendrán archivos html, para facilitar invocarlos cada vez que lo necesitemos tan sólo colocando el nombre del archivo como lo hicimos en la función del archivo **views.py**.
13. TEMPLATES = [
14. {
15. 'BACKEND': 'django.template.backends.django.DjangoTemplates',
16. 'DIRS': ['./Clases\_Coder/plantillas/'],
17. 'APP\_DIRS': True,
18. 'OPTIONS': {
19. 'context\_processors': [
20. 'django.template.context\_processors.debug',
21. 'django.template.context\_processors.request',
22. 'django.contrib.auth.context\_processors.auth',
23. 'django.contrib.messages.context\_processors.messages',
24. ],
25. },
26. },
27. ]

NOTA: en producción es recomendable colocar el path entero, pero con el punto (.) nos estamos asegurando que el repositorio va a funcionar en cualquier PC que sea descargado. Dicho punto representa todo el path hasta la carpeta donde se encuentra el archivo **manage.py**.

1. Actualizamos el path.
2. from django.contrib import admin
3. from django.urls import path
4. from .views import (
5. saludo,
6. muestra\_nombre,
7. probando\_template,
8. usando\_loader
9. )
10. urlpatterns = [
11. path('admin/', admin.site.urls),
12. path('saludo/', saludo),
13. path('muestra\_nombre/<nombre>/', muestra\_nombre),
14. path('probando\_template/', probando\_template),
15. path('usando\_loader/', usando\_loader),
16. ]

NOTA: Podemos ver que usando un loader ya no tenemos el problema de los acentos y caracteres especiales.

**Django: Modelos y Aplicaciones**

Una vez que entendemos lo que es una aplicación, vamos a crearla para luego configurar un modelo de Django.

1. Creamos la app ubicandonos desde la consola donde se encuentra el archivo **manage.py**:  
   python manage.py startapp <nombre\_app>  
   Ej: python manage.py startapp AppCoder
2. Ahora debemos avisarle a Django que hemos creado una aplicación nueva y para que la tenga mapeada la vamos a agregar a su lista de aplicaciones, en el archivo **settings.py**:
3. INSTALLED\_APPS = [
4. 'django.contrib.admin',
5. 'django.contrib.auth',
6. 'django.contrib.contenttypes',
7. 'django.contrib.sessions',
8. 'django.contrib.messages',
9. 'django.contrib.staticfiles',
10. 'AppCoder',
11. ]
12. Con el comando anterior (paso 1), se nos ha creado una carpeta (AppCoder) donde se ubicarán los archivos de la app incluyendo el modelo. En el mismo (**models.py**) vamos a crear las siguientes tablas:
13. from django.db import models
14. class Curso(models.Model):
15. nombre = models.CharField(max\_length=40)
16. camada = models.IntegerField()
17. class Estudiante(models.Model):
18. nombre = models.CharField(max\_length=40)
19. apellido = models.CharField(max\_length=20)
20. email = models.EmailField(max\_length=40)
21. class Profesor(models.Model):
22. nombre = models.CharField(max\_length=40)
23. apellido = models.CharField(max\_length=20)
24. email = models.EmailField(max\_length=40)
25. apellido = models.CharField(max\_length=30)
26. class Entregable(models.Model):
27. nombre = models.CharField(max\_length=30)
28. fecha\_de\_entrega = models.DateField()
29. entregado = models.BooleanField()
30. Django nos provee de un sistema de diagnóstico para saber si la app que venimos creando está correcta, lo ejecutamos de la siguiente manera:  
    python manage.py check AppCoder
31. Ya podemos crear los archivos de migraciones, los que le brindarán las directivas para crear las tablas en la base de datos:  
    python manage.py makemigrations  
    Dicho comando, creará en la carpeta de migraciones un archivo numerado con los comandos para que el ORM genere las tablas.

NOTA 1: Se pueden observar las verdaderas directivas SQL que se van a ejcutar con el comando: python manage.py sqlmigrate AppCoder0001  
NOTA 2: También podemos ver la lista de migraciones que Django está reconociendo y nos indica las que fueron realizadas y las que no: python manage.py showmigrations

1. Ahora podemos ejecutar esas migraciones, ya que sólo las hemos creado pero no se han ejecutado:  
   python manage.py migrate

**DbBrowser**

* En esta parte de la clase, se muestra en acción el programa realizando un CRUD. No lo incluímos en éste archivo.

**Django: Shell**

Django nos proporciona su propia shell para interactuar con el back, en el curso se muestran los pasos en vivo para agregar un registro en el modelo de **Curso**. A continuación vamos a resumir los comandos sin mucho detalle:

1. Abrimos la shell:  
   python manage.py shell
2. Importamos la tabla Curso:  
   from AppCoder.models import Curso
3. Creamos una nueva instancia, un nuevo registro:  
   curso = Curso(nombre="Python", camada=40450)
4. Guardamos los cambios, ya que sólo se encuentran en RAM:  
   curso.save()

**Django: Agregando información a la base de datos**

Desde nuestras vistas podemos agregar información a la base de datos. Por el momento vamos a seguir con el método GET, vamos a crear registros nuevos en la tabla de **Cursos** indicando tanto el nombre de una camada como la comisión.

1. Generamos una nueva vista importando previamente el modelo de Curso:
2. from AppCoder.models import Curso
3. def curso(request, nombre, numero):
4. curso = Curso(nombre=nombre, camada=int(numero))
5. curso.save()
6. documento = f"Curso: {curso.nombre}<br>Camada: {curso.camada}"
7. return HttpResponse(documento)
8. Agregamos el path nuevo importando previamente la función:
9. from django.contrib import admin
10. from django.urls import path
11. from .views import (
12. saludo,
13. muestra\_nombre,
14. probando\_template,
15. usando\_loader,
16. curso
17. )
18. urlpatterns = [
19. path('admin/', admin.site.urls),
20. path('saludo/', saludo),
21. path('muestra\_nombre/<nombre>/', muestra\_nombre),
22. path('probando\_template/', probando\_template),
23. path('usando\_loader/', usando\_loader),
24. path('curso/<nombre>/<numero>/', curso),
25. ]

**Subimos los cambios a GitHub**

1. Subimos los cambios a nuestro repositorio de GitHub:
   * git add .
   * git commit -m "Paso 1 del proyecto: 1er clase"
   * git push --set-upstream origin clase\_18-Portfolio\_Parte\_II
2. En Github realizamos un PR y hacemos el merge a **main**.

**CLASES DE DJANGO EN CODERHOUSE**

**CLASE 3: Clase 19 - Playground intermedio Parte I**

**Entorno virtual**

De ahora en más, vamos a interpretar que el entorno virtual es activado en cada clase.

**Git**

1. Nos vamos a mover a la rama main, actualizarla y crear una nueva rama para la nueva clase: git checkout main
2. Actualizamos con: git pull. Esto se descarga los cambios que hayan en la rama main. Recordemos que el PR realizado en la clase anterior agregó archivos y modificaciones en la rama main.
3. Creamos y nos movemos a la nueva rama: git checkout -b clase\_19-Playground\_intermedio\_Parte\_I

**Limpiamos vistas**

Cuando creamos el proyecto anterior, hemos creado las vistas (**views.py**) y agregamos las urls asociadas a esas vistas. Lo que no estaba correcto es crear y modificar estos archivos en la carpeta principal de configuración del proyecto, por lo tanto, eliminamos de la carpeta Clases\_Coder/Clases\_Coder/views.py y luego restauramos el archivos de **urls.py**.

from django.contrib import admin

from django.urls import path

urlpatterns = [

path('admin/', admin.site.urls),

]

A partir de ahora vamos a crear las vistas dentro de las aplicaciones.  
Para el cambio aplicado podemos hacer un commit y luego continuar.

**Django: Urls avanzadas**

Resumen:  
Por más que nuestro proyecto sea simple, rápidamente puede contenter múltiples páginas, tengamos en cuenta que una única tabla de nuestra base de datos muchas veces requiere hasta 4 páginas para poder realizar un CRUD en la misma. Si a eso sumamos index, about y más tablas, rápidamente contaremos con múltiples vistas por más que contemos con un proyecto relativamente pequeño.  
Para solucionar ésto podemos crear aplicaciones (de Django) donde no sólo contenemos nuestra base de datos, sino también podemos generar las vistas asociadas a dicha app. Asimismo, estructurar con aplicaciones nuestro proyecto nos facilita enormemente escalarlo o realizar cambios sobre el mismo en la medida en que lo necesitemos.  
**Para crear nuestras vistas en una app:**

NOTA: Continuamos el proceso trabajando sobre el proyecto actual, donde contamos con nuestra AppCoder.

1. Crear en la app **AppCoder** un archivo de **urls.py** con los path a los cuáles queremos agregar al proyecto:

from django.urls import path

from AppCoder import views

urlpatterns = [

path('/', views.inicio),

path('cursos/', views.cursos),

path('profesores/', views.profesores),

path('estudiantes/', views.estudiantes),

path('entregables/', views.entregables)

]

1. En el archivo **urls.py** que teníamos dentro de la carpeta principal de configuración del proyecto, vamos a dejar el acceso al admin y en vez de cargar un path para cada una de nuestras vistas, vamos a incluír todo el archivo **urls.py** que acabamos de crear en la app, de la siguiente manera:

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path('admin/', admin.site.urls),

path('AppCoder/', include('AppCoder.urls'))

]

1. Creamos en el archivo de **views.py** (en la app se crea automáticamente) las vistas a realizar, por el momento serán sencillas:

from django.http import HttpResponse

def inicio(request):

return HttpResponse("Vista inicio")

def cursos(request):

return HttpResponse("Vista cursos")

def profesores(request):

return HttpResponse("Vista profesores")

def estudiantes(request):

return HttpResponse("Vista estudiantes")

def entregables(request):

return HttpResponse("Vista entregables")

NOTA: Ya podemos probar su funcionamiento levantando el servidor.

1. Si no tenemos un modelo, podemos copiar el siguiente:

from django.db import models

*# Create your models here.*

class Curso(models.Model):

nombre = models.CharField(max\_length=40)

camada = models.IntegerField()

class Estudiante(models.Model):

nombre = models.CharField(max\_length=40)

apellido = models.CharField(max\_length=20)

email = models.EmailField(max\_length=40)

class Profesor(models.Model):

nombre = models.CharField(max\_length=40)

apellido = models.CharField(max\_length=20)

email = models.EmailField(max\_length=40)

apellido = models.CharField(max\_length=30)

class Entregable(models.Model):

nombre = models.CharField(max\_length=30)

fecha\_de\_entrega = models.DateField()

entregado = models.BooleanField()

**Agregamos los templates en una APP**

Resumen:  
A partir de ahora vamos a perfeccionar nuestro front haciendo uso de plantillas creadas en Bootstrap o pueden ser extraídas de otra web.  
A continuación, un "paso a paso" para ver cómo incorporar una plantilla creada en bootstrap:

1. Ingresar a la web: [Bootstrap - Templates](https://startbootstrap.com/templates)  
   Podemos buscar entre múltiples tipos de páginas, pero nosotros vamos a filtrar en las de propósito general y gratuitas.
2. Una vez en la web, seleccionamos la que más se ajuste a nuestro proyecto y hacemos clic.
3. Hacemos clic en **Free Donwnload**.
4. Una vez descargado el archivo .zip, podemos descomprimirlo y llevar sus carpetas a nuestro proyecto.
5. Para nuestro caso donde tenemos una app llamada **AppCoder**, vamos a ingresar a dicha carpeta y crear 2 nuevas carpetas: static/AppCoder y templates/AppCoder. Es decir, dentro de la carpeta de la app tenemos que crear **static** y **templates**, pero dentro de cada una de ellas vamos a crear una nueva carpeta con el mismo nombre de la app a la que pertenecen.
6. Dentro de templates/AppCoder vamos a colocar nuestros archivos .html.
7. Dentro de la carpeta static/AppCoder vamos a colocar las demás carpetas descargadas de Bootstrap, que suelen ser: assets - css - js.

Ahora lo que nos falta es indicarle a nuestras funciones en **views.py** que rendericen el html recién cargado junto a los archivos estáticos (assets, css y js):

1. Indicamos en **AppCoder/views.py** apunte a nuestro nuevo index, o sea, el **index.html**.
2. Vamos a editar el index para que Django luego sepa cargar los archivos estáticos del mismo:
   * Debemos colocar en el inicio del archivo (puede ser la primer línea dentro del ), la siguiente sintaxis: {% load static %}
   * Luego, modificamos los links o referencias a todos los archivos externos al html (css, js, img, etc), cambiando:
3. ESTO
4. <link href="css/styles.css" rel="stylesheet" />
5. POR ESTO
6. <link href="{% static 'AppCoder/css/styles.css' %}" rel="stylesheet" />
7. Ya estamos en condiciones de editar el archivo html a mano para que concuerde con nuestro proyecto.

**Subimos los cambios a GitHub**

1. Subimos los cambios a nuestro repositorio de GitHub:
   * git add .
   * git commit -m "Paso 3 del proyecto: 3er clase"
   * git push --set-upstream origin clase\_19-Playground\_intermedio\_Parte\_I
2. En Github realizamos un PR y hacemos el merge a **main**.

**CLASES DE DJANGO EN CODERHOUSE**

**CLASE 4: Clase 20 - Playground intermedio Parte II**

**Entorno virtual**

Interpretamos que el entorno virtual es activado en cada clase.

**Git**

1. Nos vamos a mover a la rama main, actualizarla y crear una nueva rama para la nueva clase: git checkout main
2. Actualizamos con: git pull. Esto se descarga los cambios que hayan en la rama main. Recordemos que el PR realizado en la clase anterior agregó archivos y modificaciones en la rama main.
3. Creamos y nos movemos a la nueva rama: git checkout -b clase\_20-Playground\_intermedio\_Parte\_II

**Herencia de Templates**

Al crear muchos archivos html podemos darnos cuenta que hay código que se repiten en todos los archivos como la navbar o el footer. Podemos aplicar **Herencia de templates** para evitar escribirlo más de una vez y centralizar el código, facilitando posibles futuros cambios:

1. Crear un archivo html que nos sirva de base, del que todos van a heredar, donde ahí colocaremos la navbar, footer y todo lo que se repita en nuestro proyecto.
2. Luego, vamos a ubicar las partes donde consideramos que cambiarían en cada página y la encerramos entre el juego de llaves y porcentajes: {% %}.  
   Por ejemplo, si queremos que el título de la pestaña cambie en función a la página que se visita, podemos escribir lo siguiente:  
   {% block title %} Index {% endblock title %}  
   De ésta forma, cada página que hereda del html base podrá ingresar su propio título o bien, si no colocamos nada en la página que hereda quedará por defecto "Index".  
   Nuestro html de base quedaría así:
3. <!DOCTYPE html>
4. <html lang="es">
5. <head>
6. {% load static %}
7. <meta charset="utf-8" />
8. <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no" />
9. <meta name="description" content="" />
10. <meta name="author" content="" />
11. <title>{% block title %} Index {% endblock title %}</title>
12. <link rel="icon" type="image/x-icon" href="assets/favicon.ico" />
13. *<!-- Font Awesome icons (free version)-->*
14. <script src="https://use.fontawesome.com/releases/v6.3.0/js/all.js" crossorigin="anonymous"></script>
15. *<!-- Core theme CSS (includes Bootstrap)-->*
16. <link href="{% static 'AppCoder/css/styles.css' %}" rel="stylesheet" />
17. </head>
18. <body id="page-top">
19. *<!-- Navigation-->*
20. <nav class="navbar navbar-expand-lg navbar-dark navbar-custom fixed-top">
21. <div class="container px-5">
22. <a class="navbar-brand" href="#page-top">Start Bootstrap</a>
23. <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarResponsive" aria-controls="navbarResponsive" aria-expanded="false" aria-label="Toggle navigation"><span class="navbar-toggler-icon"></span></button>
24. <div class="collapse navbar-collapse" id="navbarResponsive">
25. <ul class="navbar-nav ms-auto">
26. <li class="nav-item"><a class="nav-link" href="#!">Iniciar sesión</a></li>
27. <li class="nav-item"><a class="nav-link" href="#!">Crear cuenta</a></li>
28. </ul>
29. </div>
30. </div>
31. </nav>
32. {% block main %}
33. {% endblock main %}
34. *<!-- Footer-->*
35. <footer class="py-5 bg-black">
36. <div class="container px-5"><p class="m-0 text-center text-white small">Copyright &copy; Your Website 2023</p></div>
37. </footer>
38. </body>
39. </html>
40. Por último, creamos todos los html que heredan de ese archivo base:
    * Indicar que vamos a heredear del otro archivo: {% extends 'AppCoder/base.html' %}
    * También cargamos los estáticos: {% load static %}
    * Y creamos el bloque que queremos incrustar: {% block title %} Template hecho con Herencia {% endblock title %}
41. {% extends 'AppCoder/base.html' %}
42. {% load static %}
43. {% block title %} Template hecho con Herencia {% endblock title %}
44. {% block main %}
45. <h1>Este es el título del Inicio que cambio</h1>
46. <p>Se ha heredado todo desde la plantilla padre</p>
47. <h3>En el hijo, inicio.html, casí no hay nada :)</h3>
48. {% endblock main %}

**Navegando entre templates**

Para navegar en nuestro sitio, debemos utilizar una sintaxis diferente dentro de nuestros html.

1. Vamos a indicar en nuestro **urls.py** un nombre para cada url:
2. urlpatterns = [
3. path('/', views.inicio, name="Inicio"),
4. path('profesores/', views.profesores, name="Profesores"),
5. path('estudiantes/', views.estudiantes, name="Estudiantes"),
6. path('cursos/', views.cursos, name="Cursos"),
7. path('entregables/', views.entregables, name="Entregables")
8. ]
9. En los llamados a nuestros links en el html, debemos indicarlos con nueva sintaxis pero haciendo referencia a los nombres recién creados:
10. <nav class="navbar navbar-light bg-light static-top">
11. <div class="container">
12. <a class="navbar-brand" href="{% url 'Inicio' %}">Inicio</a>
13. <a class="navbar-brand" href="{% url 'Profesores' %}">Profesores</a>
14. <a class="navbar-brand" href="{% url 'Estudiantes' %}">Estudiantes</a>
15. <a class="navbar-brand" href="{% url 'Cursos' %}">Cursos</a>
16. <a class="navbar-brand" href="{% url 'Entregables' %}">Entregables</a>
17. <a class="btn btn-primary" href="#NADAAUN">INICIAR</a>
18. </div>
19. </nav>
20. En caso de que aún estemos usando el método render en las vistas, le cambiamos a todas el HttpResponse por:
21. from django.shortcuts import render
22. def inicio(request):
23. return render(request, "AppCoder/index.html")

**Panel Admin de Django**

El panel de administración de Django es una herramienta versátil y muy útil a la hora de administrar una aplicación. Desde una interfaz gráfica podemos realizar acciones del tipo CRUD en cada una de nuestras tablas y administrar usuarios.  
Para poder utilizar nuestro Panel de Django con nuestros modelos debemos seguir los siguientes pasos:

1. Ir a cada archivo **admin.py** de cada aplicación y agregamos cada modelo de la siguiente manera:
2. from django.contrib import admin
3. from .models import Profesor, Estudiante, Curso, Entregable
4. admin.site.register(Profesor)
5. admin.site.register(Estudiante)
6. admin.site.register(Curso)
7. admin.site.register(Entregable)
8. Como no existe aún un usuario en nuestro proyecto, vamos a crear un superuser desde la consola de Django:
   * python manage.py createsuperuser
   * Cargamos un user (cesar)
   * Cargamos un mail ([a@b.com](mailto:a@b.com))
   * Cargamos el password y lo repetimos (pass123)
9. Ya podemos ingresar a nuestro panel de Admin (url: localhost:8000/admin/) y luego de ingresar las credenciales recién creadas, podremos trabajar con nuestros modelos.

**Subimos los cambios a GitHub**

1. Subimos los cambios a nuestro repositorio de GitHub:
   * git add .
   * git commit -m "Agregamos Herencia e iniciamos el Admin de Django"
   * git push --set-upstream origin clase\_20-Playground\_intermedio\_Parte\_II
2. En Github realizamos un PR y hacemos el merge a **main**.