PYTHON
nivel IV

Contenido

En este nivel, se efectuarán ejercicios para el desarrollo de Aplicaciones en entorno Web a través del framework Python Django, donde se cubrirán los siguientes aspectos:

- Introducción.
- Configurar ambiente de trabajo.
- Crear proyecto.
- Crear aplicaciones y estructura de proyecto.
- Modelos y Migraciones.
- Relaciones.
- Django Shell y Querysets.
- Configurar URLs (declaración de rutas).
- Vistas.
- Modelos.
- Sistema de Plantillas.
- Configurar archivos estáticos.
- Formularios.
- Integración de Bases de Datos a través de Django.
- ORM Django.
- Servicios Web REST con Django.

Introducción.

Django es un framework de desarrollo web de código abierto, escrito en Python, que respeta el patrón de diseño conocido como Modelo–vista–template. Fue desarrollado en origen para gestionar varias páginas orientadas a noticias de la World Company de Lawrence, Kansas, y fue liberada al público bajo una licencia BSD en julio de 2005; el framework fue nombrado en alusión al guitarrista de jazz gitano Django Reinhardt. En junio de 2008 fue anunciado que la recién formada Django Software Foundation se haría cargo de Django en el futuro.

Django se usó en producción durante un tiempo antes de que se liberara al público; fue desarrollado por Adrian Holovaty, Simon Willison, Jacob Kaplan-Moss y Wilson Miner mientras trabajaban en World Online, y originalmente se utilizó para administrar tres sitios web de noticias: The Lawrence Journal-World, lawrence.com y KUsports.com.

Preparación de ambiente

CREACIÓN DEL AMBIENTE VIRTUAL

(1) INSTALAR PYTHON 3.7 CON ACCESO A TRAVÉS DEL PATH (NO USAR LA INSTRUCCIÓN DE ABAJO)

C:\>set path=%path%;C:\Program Files\Python37;C:\Program Files\Python37\scripts

(2) #INSTALAR VIRTUALENV

C:\>pip install virtualenv --user

(3) #Añadir al path la siguiente ruta así:

C:\Users\hduqu\AppData\Roaming\Python\Python37\Scripts

set path=%path%;C:\Users\hduqu\AppData\Roaming\Python\Python37\Scripts

(4) #CREAR EL AMBIENTE VIRTUAL

C:\>virtualenv TESTDJ

(5) #SE ACTIVA EL ENTORNO VIRTUAL

>\testdj\scripts\activate

Preparación de ambiente (continuación)

(6) #INSTALE DJANGO

>pip install django

(7)# ACCEDA A PYTHON DESDE SU ENTORNO VIRTUAL

(TESTDJ) C:\>python

(8) # VERIFIQUE LOS SIGUIENTES COMANDOS:

(TESTDJ) C:\>python

Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20) [MSC v.1916 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

>>> import django

>>> print (django.get_version())

2.2.4

>>>

O TAMBIÉN PUEDE SER:

>>> django.VERSION

(2, 2, 4, 'final', 0)

Preparación de ambiente (continuación)

(9) #SE DESACTIVA EL ENTORNO VIRTUAL

>deactivate

Creación de un proyecto.

Una vez que has instalado Python, Django y (opcionalmente) una base de datos (incluyendo los controladores), puedes empezar a dar tus primeros pasos en el desarrollo de aplicaciones, creando un proyecto.

Un proyecto es una colección de configuraciones para una instancia de Django, incluyendo configuración de base de datos, opciones específicas de Django y configuraciones específicas de aplicaciones.

Creación de un proyecto.

Desde el entorno virtual, acceder al directorio TESTDJ de la siguiente forma:

(TESTDJ) C:\>cd testdj

(TESTDJ) C:\TESTDJ>

Ejecutar el comando django-admin startproject misitio, de la siguiente forma:

(TESTDJ) C:\TESTDJ>django-admin startproject misitio

Se puede apreciar como se crea el directorio \misitio

Creación de un proyecto.

El cual contendrá la siguiente estructura:

```
misitio/
    manage.py
    misitio/
        __init__.py
        settings.py
        urls.py
        wsgi.py
```

Estos archivos son los siguientes:

- **misitio/:** El directorio de trabajo externo misitio/, es solo un contenedor, es decir una carpeta que contiene nuestro proyecto. Por lo que se le puede cambiar el nombre en cualquier momento sin afectar el proyecto en sí.
- **manage.py:** Una utilidad de línea de comandos que te permite interactuar con un proyecto Django de varias formas. Usa manage.py help para ver lo que puede hacer. No deberías editar este archivo, ya que este es creado en el directorio convenientemente para manejar el proyecto.

Creación de un proyecto.

Acceder al directorio misitio , de la siguiente forma:

TESTDJ) C:\TESTDJ>cd misitio

(TESTDJ) C:\TESTDJ\misitio>dir
 Volume in drive C is Windows
 Volume Serial Number is 8E38-6A39

 Directory of C:\TESTDJ\misitio

18/08/2019 05:36 p. m.   <DIR>   .
18/08/2019 05:36 p. m.   <DIR>   ..
18/08/2019 05:36 p. m.   648 manage.py
18/08/2019 05:36 p. m.   <DIR>   misitio
               1 File(s)   648 bytes
               3 Dir(s) 812.883.181.568 bytes free

(TESTDJ) C:\TESTDJ\misitio>manage.py help

Creación de un proyecto.

Observará un resultado como el siguiente:

Type 'manage.py help <subcommand>' for help on a specific subcommand.

Available subcommands:

[auth]
    changepassword
    createsuperuser

[contenttypes]
    remove_stale_contenttypes

[django]
    check

POR EJEMPLO:

(TESTDJ) C:\TESTDJ\misitio>manage.py help runserver

Creación de un proyecto.

- **misitio/misitio/:** El directorio interno misitio/ contiene el paquete Python para tu proyecto. El nombre de este paquete Python se usará para importar cualquier cosa dentro del proyecto. (Por ejemplo import misitio.settings).

- **__init__.py:** Un archivo requerido para que Python trate el directorio misitio como un paquete o como un grupo de módulos. Es un archivo vacío y generalmente no necesitaras agregarle nada.

- **settings.py:** Las opciones/configuraciones para nuestro proyecto Django. Dale un vistazo, para que te des una idea de los tipos de configuraciones disponibles y sus valores predefinidos.

- **urls.py:** Declaración de las URLs para este proyecto de Django. Piensa que es como una ‘‘tabla de contenidos’’ de tu sitio hecho con Django.

- **wsgi.py**: El punto de entrada WSGI para el servidor Web, encargado de servir nuestro proyecto.

Creación de un proyecto.

Todos estos pequeños archivos, constituyen un proyecto Django, que puede albergar múltiples aplicaciones. Observa que la variable INSTALLED_APPS, hacia el final del archivo settings.py, contiene el nombre de todas las aplicaciones Django que están activadas en esta instancia de Django. Las aplicaciones pueden ser empacadas y distribuidas para ser usadas por otros proyectos.

De forma predeterminada INSTALLED_APPS contiene todas las aplicaciones, que vienen por defecto con Django:

- django.contrib.admin -- La interfaz administrativa.
- django.contrib.auth -- El sistema de autentificación.
- django.contrib.contenttypes -- Un framework para tipos de contenidos.
- django.contrib.sessions -- Un framework. para manejar sesiones
- django.contrib.messages -- Un framework para manejar mensajes
- django.contrib.staticfiles -- Un framework para manejar archivos estáticos.

Estas aplicaciones se incluyen por defecto, como conveniencia para los casos más comunes.

Creación de un proyecto.

- Algunas de estas aplicaciones hacen uso, de por lo menos una tabla de la base de datos, por lo que necesitas crear las tablas en la base de datos, antes de que puedas utilizarlas, para hacerlo entra al directorio que contiene tu proyecto (cd misitio) y ejecuta el comando siguiente, para activar el proyecto Django.: python manage.py migrate

(TESTDJ) C:\TESTDJ\misitio>python manage.py migrate

Creación de un proyecto.

- El comando migrate busca la variable *INSTALLED_APPS* y crea las tablas necesarias de cada una de las aplicaciones registradas en el archivo settings.py, que contiene todas las aplicaciones. Veras un mensaje por cada migración aplicada.

```
(TESTDJ) C:\TESTDJ\misitio>python manage.py migrate
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 sessions.0001_initial... OK
```

Crear aplicaciones y estructura de proyecto

Para obtener un poco de información y más retroalimentación, ejecuta el servidor de desarrollo de Django, para ver el proyecto en acción.

Django incluye un servidor Web ligero (Que es llamado con el comando ‘‘runserver’’) que puedes usar mientras estás desarrollando tu sitio. Se incluye este servidor para que puedas desarrollar tu sitio rápidamente, sin tener que lidiar con configuraciones de servidores Web para producción (por ejemplo, Apache).

Este servidor de desarrollo vigila tu código a la espera de cambios y se reinicia automáticamente, ayudándote a hacer algunos cambios rápidos en tu proyecto sin necesidad de reiniciar nada.

Para iniciar el servidor, entra en el directorio que contiene tu proyecto (cd misitio) y ejecuta el comando:

python manage.py runserver

Crear aplicaciones y estructura de proyecto

El resultado a obtener, sería el siguiente:

```
(TESTDJ) C:\TESTDJ\misitio>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
August 18, 2019 - 18:31:10
Django version 2.2.4, using settings 'misitio.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Crear aplicaciones y estructura de proyecto

- **Cambiar el host y el puerto**

Por defecto, el comando runserver inicia el servidor de desarrollo en el puerto 8000, escuchando sólo conexiones locales. Si quieres cambiar el puerto del servidor, pásalo a este como un argumento en la línea de comandos:

python manage.py runserver 8080

Por ejemplo:

```
(TESTDJ) C:\TESTDJ\misitio>python manage.py runserver 8080
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
August 18, 2019 - 18:37:03
Django version 2.2.4, using settings 'misitio.settings'
Starting development server at http://127.0.0.1:8080/
Quit the server with CTRL-BREAK.
```

Crear aplicaciones y estructura de proyecto

- **La dirección IP también puede ser modificada, de la siguiente forma:**

```
>python manage.py runserver 192.148.1.103:8000
```

Vista creada con Django

- Crear el archivo views.py, dentro del directorio \misitio (el cual había sido creado en un inicio con el comando: django-admin.py startproject misitio), al mismo nivel donde se encuentra settings.py

Contenido views.py:

```
from django.http import HttpResponse
def hola(request):
        return HttpResponse("Hola estimados alumnos de UNEWEB")
```

Vista creada con Django

Otro ejemplo: Modificar el archivo views.py, con el siguiente contenido:

```
from django.http import HttpResponse
HTML = """
    <!DOCTYPE html>
    <html lang="es">
    <head>
    <meta httpequiv="
    contenttype"
    content="text/html; charset=utf8">
    <meta name="robots" content="NONE,NOARCHIVE">
    <title>Hola alumnos UNEWEB</title>
```

Vista creada con Django

Otro ejemplo: Modificar el archivo views.py, con el siguiente contenido (continuación):

```
<style type="text/css">
html * { padding:0; margin:0; }
body * { padding:10px 20px; }
body * * { padding:0; }
body { font:small sansserif;
}
body>div { borderbottom:
1px solid #ddd; }
h1 { fontweight:
normal; }
```

Vista creada con Django

Otro ejemplo: Modificar el archivo views.py, con el siguiente contenido (continuación):

```
#summary { background: #e0ebff; }
</style>
</head>
<body>
<div id="summary">
<h1>¡Hola estimados alumnos de UNEWEB. Habla Django!</h1>
</div>
</body></html> """
def hola(request):
        return HttpResponse(HTML)
```

Vista creada con Django

Una vista es solo una función Python, que toma como primer argumento una petición HttpRequest y retorna como respuesta una instancia de HttpResponse. Por lo que una función en Python es una vista en Django.

URLconf creada con Django

Una URLconf es como una tabla de contenidos para tu sitio web hecho con Django. Básicamente, es un mapeo entre los patrones URL y las funciones de vista que deben ser llamadas por esos patrones URL. Es como decirle a Django, ‘‘Para esta URL, llama a este código, y para esta otra URL, llama a este otro código’’. Por ejemplo, ‘‘Cuando alguien visita la URL /hola/, llama a la función vista_hola() la cual está en el modulo Python views.py.’’

URLconf creada con Django

Modificar el archivo urls.py, contenido en el directorio \misitio, al mismo nivel del archivo settings.py. Añadir el siguiente contenido:

urls.py

```
from django.urls import path
from misitio.views import hola

urlpatterns = [
    path('hola/',hola),
]
```

URLconf creada con Django

Contenido:

- La primera línea importa las funciones: path e include, del modulo django.conf.urls, la función path es una tupla, donde el primer elemento es una ruta y el segundo elemento es la función de vista que se usa para ese enlace, mientras que la función include se encarga de importar módulos que contienen otras URLconf, al camino de búsqueda de Python, como una forma de ‘‘incluir’’ urls que pertenecen a otro paquete, en este caso al sitio administrativo, que viene activado por .

URLconf creada con Django

Contenido:

- Después tenemos a la función urlpatterns(), una variable que recibe los argumentos de las url en forma de lista, inclusive cadenas de caracteres vacías.
- Por defecto, todo lo que está en la URLconf está comentado e incluye algunos ejemplos de configuraciones comúnmente usados, a excepción del sitio administrativo, el cual esta activado por omisión. (Para desactivarlo, solo es necesario comentarlo.)

URLconf creada con Django

Contenido:

- Primero, importamos la vista hola, desde el modulo misitio/views.py que en la sintaxis de import de Python se traduce a misitio.views. (La cual asume que el paquete misitio/views.py, está en la ruta de búsqueda de Python o python path).
- Luego, agregamos la línea path('hola/', hola), a urlpatterns Esta línea hace referencia a un URLpattern -- Una tupla de Python en dónde el primer elemento es una ruta y el segundo elemento es la función de vista que usa para manejar ese enlace.

URLconf creada con Django

Contenido:

- Si ignoramos el código comentado y las referencias a la interfaz administrativa, esto es esencialmente una URLconf:

```
from django.urls import path

urlpatterns = [

]
```

URLconf creada con Django

Contenido: (Respecto a las rutas)

- La ruta 'hola/' significa que es una cadena de caracteres en crudo de Python.
- Puedes excluir la barra al comienzo de la expresión 'hola/' para que coincida con /hola/. Django automáticamente agrega una barra antes de toda expresión. A primera vista esto parece raro, pero una URLconf puede ser incluida en otra URLconf, y el dejar la barra de lado simplifica mucho las cosas.

URLconf creada con Django

Otro ejemplo de rutas:

```
from django.urls import path
from misitio.views import raiz, hola
urlpatterns = [
        path('raiz/', raiz),
        path('hola/', hola),
# ...
]
```

URLconf creada con Django

Prueba: Modificar el archivo view.py, con el siguiente contenido:

```
from django.http import HttpResponse
HTML = """
…
def hola(request):
        return HttpResponse(HTML)
def raiz(request):
        return HttpResponse('<h1>Hola estoy en la raiz</h1>')
```

URLconf creada con Django

En resumen, el algoritmo sigue los siguientes pasos:

1. Se recibe una petición, por ejemplo a /hola/
2. Django determina la URLconf a usar, buscando la variable ROOT_URLCONF en el archivo de configuraciones.
3. Django busca todos los patrones en la URLconf buscando la primera coincidencia con /hola/.
4. Si encuentra uno que coincida, llama a la función vista asociada.
5. La función vista retorna una HttpResponse.
6. Django convierte el HttpResponse en una apropiada respuesta HTTP, la cual convierte en una página Web.

Path ambiente de trabajo

```
>>> from __future__ import print_function
>>> import sys
>>> print (sys.path)
```

Otro ejemplo de vistas y urls

```
>>> from __future__ import print_function
>>> import datetime
>>> ahora = datetime.datetime.now()
>>> ahora
datetime.datetime(2019, 8, 19, 6, 28, 32, 9639)
>>> print(ahora)
2019-08-19 06:28:32.009639
>>>
```

Otro ejemplo de vistas y urls

- Para crear esta página, crearemos una función de vista, que muestre la fecha y la hora actual, por lo que necesitamos anclar la declaración datetime.datetime.now() dentro de la vista para que la retorne como una respuesta HttpResponse. Esta es la vista que retorna la fecha y hora actual, como un documento HTML. Así como la función hola, que creamos en la vista anterior, la función fecha_actual debe de colocarse en el mismo archivo views.py.

```
from django.http import HttpResponse
import datetime
def fecha_actual(request):
        ahora = datetime.datetime.now()
        html =
“<html><body><h1>Fecha:</h1><h3>%s<h/3></body></html>"  %
ahora
        return HttpResponse(html)
```

Otro ejemplo de vistas y urls

- Se actualiza el archivo urls.py, con el siguiente contenido:

```
from django.conf.urls import path
from misitio.views import hola, hola1, hola2, fecha_actual, raiz, otra_fecha
urlpatterns = [
   #path('admin/', admin.site.urls),
    path('hola/',hola),
    path('hola1/',hola1),
    path('hola2/',hola2),
    path('raiz/',raiz),
    path('otra_fecha/', fecha_actual)
]
```

Otro ejemplo de vistas y urls

- **Se actualiza el archivo views.py, con el siguiente contenido:**

```
from django.http import HttpResponse
import datetime
HTML = """
<!DOCTYPE html>
…
<h1>¡Hola estimados alumnos de UNEWEB. Habla Django!</h1>
</body></html> """
def hola(request):
        return HttpResponse(HTML)
def raiz(request):
        return HttpResponse('<h1>Hola estoy en la raiz</h1>')
def fecha_actual(request):
        ahora = datetime.datetime.now()
        html =
        "<html><body><h1>Fecha:</h1><h3>%s<h/3></body></html>"%ahora
        return HttpResponse(html)
```

Otro ejemplo de vistas y urls

- **Se actualiza el archivo views.py, con el siguiente contenido:**

```
from django.http import Http404, HttpResponse
import datetime
…
def horas_adelante(request,offset):
        try:
                offset = int(offset)
        except valueError:
                raise Http404()
        dt = datetime.datetime.now()+datetime.timedelta(hours=offset)
        html = "<h1>En %s hora(s), serán:<h3>%s</h3>"%(offset,dt)
        return HttpResponse(html)
```

Otro ejemplo de vistas y urls

- **Se actualiza el archivo urls.py, con el siguiente contenido:**

```
from django.urls import path
from django.urls import re_path

from misitio.views import hola, hola1, hola2, fecha_actual, raiz, otra_fecha, horas_adelante

urlpatterns = [
    path('admin/', admin.site.urls),
    path('hola/',hola),
    path('hola1/',hola1),
    path('hola2/',hola2),
    path('raiz/',raiz),
    path('otra_fecha/', fecha_actual),
    re_path(r'^fecha/mas/(\d{1,2})/$', horas_adelante)
]
```

El sistema de plantillas

Esta es la forma básica, en la que podemos usar el sistema de plantillas de Django en código Python.

1. Crea un objeto Template pasándole el código en crudo de la plantilla como una cadena.

2. Llama al método render() del objeto Template con un conjunto de variables (o sea, el contexto). Este retorna una plantilla totalmente renderizada como una cadena de caracteres, con todas las variables y etiquetas de bloques evaluadas de acuerdo al contexto.

Usando código, esta es la forma que podría verse, solo inicia el intérprete interactivo con:

python manage.py shell

El sistema de plantillas

```
(TESTDJ) C:\TESTDJ\misitio>python manage.py shell
…
(InteractiveConsole)
>>> from __future__ import print_function
>>> from django import template
>>> t = template.Template('Mi nombre es:{{nombre}}.')
>>> c = template.Context({'nombre':'Jose'})
>>> print (t.render(c))
Mi nombre es:Jose.
>>> c = template.Context({'nombre':'Juan'})
>>> print (t.render(c))
Mi nombre es:Juan.
>>>
```

El sistema de plantillas

Crear el directorio: **templates** y el archivo: **fecha_actual.html**. Debe quedar de la siguiente forma:

misitio/misitio/templates/fecha_actual.html

Contenido: **fecha_actual_nueva.html**

El sistema de plantillas

```
{% load static %}
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="{% static 'fecha_actual_nueva.css' %}">
    <title>Document</title>
</head>
<body>
    <h1>Fecha actual</h1>
    <h3>Hoy es {{ fecha_actual_tmp }}</h3>
</body>
</html>
```

El sistema de plantillas

Contenido del archivo “fecha_actual_nueva.css”

```
*{
   font-family: Arial, Helvetica, sans-serif;
   text-align: center;
}
p{
   background-color: #ccc;
   padding: 20px;
   width: 50%;
   margin-left: 25%;
}
```

El sistema de plantillas

- Modificar el archivo **settings.py**, incorporando el siguiente contenido:

```
import os.path
```

- En TEMPLATES

```
'DIRS': [os.path.join(os.path.dirname(__file__),
'templates').replace('\\','/'),],

STATIC_URL = '/static/‘

STATICFILES_DIRS = [
    BASE_DIR / "misitio/static",
]
```

El sistema de plantillas

Modificar el archivo **views.py**, dejando sólo el siguiente contenido:

```
import datetime
from django.shortcuts import render
def fecha_actual_nueva(request):
    ahora = datetime.datetime.now()
    return render(request, 'fecha_actual_nueva.html',
{'fecha_actual_tmp':ahora})
```

El sistema de plantillas

El método render()

El primer argumento de render() debe ser el nombre de la plantilla a utilizar. El segundo argumento, si es pasado, debe ser un diccionario para usar en la creación de un Context para esa plantilla. Si no se le pasa un segundo argumento, render utilizará un diccionario vacío.

El sistema de plantillas

Por último modificar el archivo urls.py, dejar el siguiente contenido y probar:

```
from django.conf.urls import url
from misitio.views import fecha_actual
urlpatterns = [
    path('admin/', admin.site.urls),
    path('hola/',hola),
    path('hola1/',hola1),
    path('hola2/',hola2),
    path('raiz/',raiz),
    path('otra_fecha/',fecha_actual),
    re_path(r'^fecha/mas/(\d{1,2})/$',horas_adelante),
    path('fecha_actual_template/',fecha_actual_nueva)
]
```

Probar:

http://127.0.0.1:8000/fecha_actual_template/

El sistema de plantillas

La etiqueta de plantilla include.

Ahora que hemos visto en funcionamiento el mecanismo para cargar plantillas, podemos introducir un tipo de plantilla incorporada que tiene una ventaja para esto:

{% include %}. Esta etiqueta te permite incluir el contenido de otra plantilla. Por ejemplo:

El sistema de plantillas

Crear dentro del directorio \templates, el directorio \includes. Crear dentro de este directorio el archivo: nav.html, con el siguiente contenido:

```
<div id="nav">
        Tu estas en: {{ seccion_actual }}
</div>
```

El sistema de plantillas

Ahora modificar el archivo views.py, con el siguiente contenido:

```
import datetime
from django.shortcuts import render
def fecha_actual_nueva_include(request):
    ahora = datetime.datetime.now()
    return render(request,
'fecha_actual_nueva_include.html',
{'fecha_actual_tmp':ahora, 'seccion_actual':'INCLUIR'})
```

El sistema de plantillas

Por último modificar el archivo fecha_actual_nueva_include.html, con el siguiente contenido y probar:

```
{% load static %}

<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Document</title>
</head>
<body>
   <h1 style='text-align:center;'>Fecha actual</h1>

   {% include "includes/nav.html" %}

   <h3 style='text-align:center;'>Hoy es {{fecha_actual_tmp}}</h3>
</body>
</html>
```

Probar: http://127.0.0.1:8000/fecha_actual_include/

MVC

Django sigue el patrón MVC tan al pie de la letra que puede ser llamado un framework MVC. Someramente, la M, V y C se separan en Django de la siguiente manera:

- M, la porción de acceso a la base de datos, es manejada por la capa de la base de datos de Django.
- V, la porción que selecciona qué datos mostrar y cómo mostrarlos, es manejada por la vista y las plantillas.
- C, la porción que delega a la vista dependiendo de la entrada del usuario, es manejada por el framework mismo siguiendo tu URLconf y llamando a la función apropiada de Python para la URL obtenida.

MVT

Debido a que la ‘‘C’’ es manejada por el mismo framework y la parte más compleja se produce en los modelos, las plantillas y las vistas, Django es conocido como un Framework MTV. En el patrón de diseño MTV.

- M significa ‘‘Model’’ (Modelo), la capa de acceso a la base de datos. Esta capa contiene toda la información sobre los datos: cómo acceder a estos, cómo validarlos, cuál es el comportamiento que tiene, y las relaciones entre los datos.
- T significa ‘‘Template’’ (Plantilla), la capa de presentación. Esta capa contiene las decisiones relacionadas a la presentación: como algunas cosas son mostradas sobre una página web o otro tipo de documento.
- V significa ‘‘View’’ (Vista), la capa de la lógica de negocios. Esta capa contiene la lógica que accede al modelo y la delega a la plantilla apropiada: puedes pensar en esto como un puente entre los modelos y las plantillas.

Modelo

- Abrir el archivo setting.py, ubicar el parámetro DATABASES:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Modelo

Nota: Cualquiera que sea la base de datos que uses, necesitarás descargar e instalar el adaptador apropiado. Cada uno de estos está disponible libremente en la web; sólo sigue el enlace en la columna ‘‘Adaptador requerido’’.

```
(testdj) F:\testdj\misitio>pip install mysql-connector --force
Collecting mysql-connector
  Using cached mysql_connector-2.2.9-cp311-cp311-win_amd64.whl
Installing collected packages: mysql-connector
  Attempting uninstall: mysql-connector
    Found existing installation: mysql-connector 2.2.9
    Uninstalling mysql-connector-2.2.9:
      Successfully uninstalled mysql-connector-2.2.9
Successfully installed mysql-connector-2.2.9
```

Modelo

La variable DATABASES en el archivo settings.py, es un diccionario que contendrá los ajustes necesarios, para configurar la base datos:

- **ENGINE = ''**
- **NAME = ''**
- **USER = ''**
- **PASSWORD = ''**
- **HOST = ''**
- **DATABASE_PORT = ''**

Modelo

- ENGINE: le indica a Django qué base de datos utilizar. Si usas una base de datos con Django, ENGINE debe configurarse con una cadena de los mostradas en la siguiente tabla:

Configuración	Base de datos	Adaptador requerido
django.db.backends.postgresql_psycopg2	PostgreSQL	Psycopg version 2.x, http://www.djangoproject.com/r/python-pgsql/.
django.db.backends.mysql	MySQL	MySQLdb, http://www.djangoproject.com/r/python-mysql/.
django.db.backends.sqlite3	SQLite	No necesita adaptador
django.db.backends.oracle	Oracle	cx_Oracle, http://www.djangoproject.com/r/python-oracle/.

Modelo

NAME la indica a Django el nombre de tu base de datos. Si estás usando SQLite, especifica la ruta completo del sistema de archivos hacia el archivo de la base de datos (por ej. '/home/django/datos.db').

USER le indica a Django cual es el nombre de usuario a usar cuando se conecte con tu base de datos. Si estás usando SQLite, deja este en blanco.

PASSWORD le indica a Django cual es la contraseña a utilizar cuando se conecte *con tu base de datos. Si estás utilizando SQLite o tienes una contraseña vacía, deja este en blanco.*

HOST *le indica a Django cual es el host a usar cuando se conecta a tu base de datos. Si tu base de datos está sobre la misma computadora que la instalación de Django (o sea localhost), deja este en blanco. Si estás usando SQLite, deja este en blanco.*

Modelo

Configuración por omisión:

La variable DATABASES, por omisión usa la configuración más simple posible, la cual está configurada para utilizar SQLite, por lo que no tendrás que configurar nada, si vas a usar SQLite como base de datos:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Modelo

Configuración específica, según sea el caso:

Sin embargo si quieres usar otra base de datos como MySQL, Oracle, o PostgreSQL es necesario especificar algunos parámetros adicionales, que serán requeridos en el archivo de configuración. El siguiente ejemplo asume que quieres utilizar MySQL:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'testdj',
        'USER': 'root',
        'PASSWORD': '',
        'HOST': 'localhost',
        'PORT': '3306',
    }
}
```

Modelo

Configuración específica, según sea el caso:

Como podrás darte cuenta, lo único que necesitas cambiar es la 'ENGINE', si quieres usar MySQL e introducir los datos apropiados de acuerdo a la base de datos que estés usando.

Una vez que hayas ingresado estas configuraciones, compruébalas. Primero, desde el directorio del proyecto que creaste. Ejecuta el comando:

python manage.py shell

Modelo

Configuración específica, según sea el caso:

- Notarás que comienza un intérprete interactivo de Python. Las apariencias pueden engañar. Hay una diferencia importante entre ejecutar el comando manage.py shell dentro del directorio del proyecto de Django y el intérprete genérico python.
- El último es el Python shell básico, pero el anterior le indica a Django cuales archivos de configuración usar antes de comenzar el shell.
- Este es un requerimiento clave para realizar consultas a la base de datos: Django necesita saber cuáles son los archivos de configuraciones a usar para obtener la información de la conexión a la base de datos.
- Detrás de escena, manage.py shell simplemente asume que tu archivo de configuración está en el mismo directorio que manage.py.

Modelo

Configuración específica, según sea el caso:

Modificar el archivo settings.py para incluir el siguiente contenido:

```
DATABASES = {
    'default': {
        'ENGINE': 'mysql.connector.django',
        'NAME': ‘testdj',
        'USER': 'root',
        'PASSWORD': '',
        'HOST': 'localhost',
    }
}
```

Modelo

Configuración específica, según sea el caso:

Ejecutar el siguiente comando:

>python manage.py shell

Si no se presenta ningún error, la configuración está bien. Una vez que hayas entrado al shell, escribe estos comandos para probar la configuración de tu base de datos:

```
>>> from django.db import connection
>>> cursor = connection.cursor()
>>>
```

Modelo

Configuración específica, según sea el caso:

Una vez creada la base de datos “testdj” en MySQL, puede ejecutar los siguientes comandos desde el shell del entorno virtual.

```
>python manage.py shell
>>> from django.db import connection
>>> connection.connect()
>>> cursor = connection.cursor()
>>> cursor.execute("SHOW DATABASES LIKE '%testdj%'")
1
>>> databases = cursor.fetchall()
>>> for database in databases:
...     print(database)
...
('testdj',)
>>>
```

Una nueva aplicación

Ahora proceda a ejecutar el siguiente comando para crear una nueva aplicación a la que llamaremos biblioteca:

- Un proyecto es una instancia de un cierto conjunto de aplicaciones de Django, más las configuraciones de esas aplicaciones. Técnicamente, el único requerimiento de un proyecto es que este suministre un archivo de configuración o settings.py, el cual define la información hacia la conexión a la base de datos, la lista de las aplicaciones instaladas, la variable TEMPLATE_DIRS, y así sucesivamente.
- Una aplicación es un conjunto portable de alguna funcionalidad de Django, típicamente incluye modelos y vistas, que conviven en un solo paquete de Python (Aunque el único requerimiento es que contenga una archivo models.py).
- Ejecutar el siguiente comando, dentro del directorio:

>python manage.py startapp biblioteca

Una nueva aplicación

Este será el contenido:

```
biblioteca/
        __init__.py
        admin.py
        models.py
        tests.py
        views.py
        migrations/
                __init__.py
```

El modelo

- El primer paso para utilizar esta configuración de base de datos con Django es expresarla como código Python. En el archivo models.py que se creó con el comando startapp, ingresa lo siguiente: biblioteca/models.py

```
from django.db import models

class Persona(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

El modelo

Modificar el archivo: /misitio/settings.py, con el siguiente contenido:

configuración termine viéndose así:

```
INSTALLED_APPS = (
'django.contrib.admin',
'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',
'biblioteca',
)
```

El modelo

Ejecutar el comando:

(TESTDJ) C:\TESTDJ\misitio>**python manage.py check biblioteca**

El comando check verifica que todo esté en orden respecto a tus modelos, no crea ni toca de ninguna forma tu base de datos -- sólo imprime una salida en la pantalla en la que identifica posibles errores en tus modelos.

El modelo

Una vez que todo está en orden, necesitamos guardar las migraciones para los modelos en un archivo de control, para que Django pueda encontrarlas al sincronizar el esquema de la base de datos. Ejecuta el comando makemigrations de esta manera:

(TESTDJ) C:\TESTDJ\misitio>**python manage.py makemigrations**

Migrations for 'biblioteca':

biblioteca\migrations\0001_initial.py

- Create model Persona

El modelo

Una vez que usamos el comando makemigrations, para crear las "migraciones", podemos usar el comando sqlmigrate para ver el SQL generado. El comando sqlmigrate toma los nombres de las migraciones y las retorna en un lenguaje SQL, por cada aplicación especificada, de la siguiente forma:

>python manage.py sqlmigrate biblioteca 0001

El modelo

>python manage.py migrate

En este comando, biblioteca es el nombre de la aplicación y 0001, es el número que Django asigna como nombre a cada migración o cambio hecho al esquema de la base de datos (revisar dentro de la carpeta migrations).

El comando migrate es una simple sincronización de tus modelos hacia tu base de datos. Este comando examina todos los modelos en cada aplicación que figure en tu

variable de configuración INSTALLED_APPS, verifica la base de datos para ver si las tablas apropiadas ya existen, y las crea si no existen

El modelo

Los tres pasos que seguimos para crear cambios en el modelo.

- 1. Cambia tu modelo (en models.py).
- 2. Ejecuta python manage.py makemigrations para crear las migraciones para esos cambios.
- 3. Ejecuta python manage.py migrate para aplicar esos cambios a la base de datos.

El modelo

Para ver opciones de campo y tipos de datos:

https://docs.djangoproject.com/en/2.2/ref/models/fields/#model-field-types

Las Migraciones

En este punto, quizás te preguntes ¿Que son las migraciones? Las migraciones son la forma en que Django se encarga de guardar los cambios que realizamos a los modelos (Agregando un campo, una tabla o borrando un modelo... etc.) en el esquema de la base de datos. Están diseñadas para funcionar en su mayor parte de forma automática, utilizan una versión de control para almacenar los cambios realizados a los modelos y son guardadas en un archivo del disco llamado ‘‘migration files’’, que no es otra cosa más que archivos Python, por lo que están disponibles en cualquier momento.

Las Migraciones

Existen dos comandos para usar e interactuar con las migraciones:

- **makemigrations**: es responsable de crear nuevas migraciones basadas en los cambios aplicados a nuestros modelos.
- **migrate**: responsable de aplicar las migraciones y los cambios al esquema de la base de datos.

Estos dos comandos se usan de forma interactiva, primero se crean o graban las migraciones, después se aplican:

python manage.py makemigrations

python manage.py migrate

Las migraciones se derivan enteramente de los archivos de los modelos y son esencialmente registros, que se guardan como historia, para que Django (o cualquier desarrollador) pueda consultarlos, cuando necesita actualizar el esquema de la base de datos para que los modelos coincidan con los modelos actuales.

Acceso a datos

Ejecutar:

(TESTDJ) C:\TESTDJ\misitio>python manage.py shell

>>> from biblioteca.models import Persona

>>> p1 = Persona(nombre='Jose',apellido='Perez')

>>> p1.save()

Otra manera equivalente a la anterior es:

>>> P2 = Persona.objects.create(nombre='Victor',apellido='Sanchez')

>>> Lista_Persona = Persona.objects.all()

>>> Lista_Persona

Posteriormente proceda a consultar la tabla Persona, a través de la interface disponible a tales efectos en el manejador de Bases de Datos de su elección.

Acceso a datos

Ejecutar:

```
>>> Lista_Personas = Persona.objects.all()

>>> Lista_Personas
<QuerySet [<Persona: Persona object (1)>]>
```

Para mayor información:

https://docs.djangoproject.com/en/2.2/topics/db/queries/

Acceso a datos

Modificar el contenido de models.py:

```
from django.db import models


class Persona(models.Model):
    nombre = models.CharField(max_length=30)
    apellido = models.CharField(max_length=30)

    def __str__(self): # __unicode__ en Python 2
        return '%s %s' % (self.nombre, self.apellido)
```

QuerySet

¿Qué es un QuerySet?

- Un QuerySet es, en esencia, una lista de objetos de un modelo determinado. Un QuerySet te permite leer los datos de la base de datos, filtrarlos y ordenarlos.

Acceso a datos (QuerySets)

Acceder a **>python manage.py shell**

Ejecutar:

```
>>> from biblioteca.models import Persona
>>> Lista_Personas = Persona.objects.all()
>>> Lista_Personas.filter(nombre='Jose')
<QuerySet [<Persona: Jose Perez>]>
>>> Lista_Personas
<QuerySet [<Persona: Jose Perez>]>
```

ORM Django

Mapeo objeto-relacional

El **mapeo objeto-relacional** (más conocido por su nombre en inglés, Object-Relational mapping, o sus siglas O/RM, ORM, y O/R mapping) es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y la utilización de una base de datos relacional como motor de persistencia. En la práctica esto crea una base de datos orientada a objetos virtual, sobre la base de datos relacional.

Acceso a datos (QuerySets/ORM)

Seleccionar Objetos:

```
>>> from biblioteca.models import Persona
>>> Persona.objects.all()
<QuerySet [<Persona: Jose Perez>]>
```

Filtrar Datos:

```
>>> Persona.objects.filter(nombre='Jose')
<QuerySet [<Persona: Jose Perez>]>
>>>
```

Obtener Objetos Individuales:

```
>>> Persona.objects.get(nombre='Jose')
<Persona: Jose Perez>
```

Acceso a datos (QuerySets/ORM)

Ordenar datos :

```
>>> p1 = Persona(nombre='Laura',apellido='Rodriguez')
>>> p1.save()
>>> p1 = Persona(nombre='Antonio',apellido='Suarez')
>>> p1.save()
```

Ascendente

```
>>> Persona.objects.order_by('nombre')
<QuerySet [<Persona: Antonio Suarez>, <Persona: Jose Perez>, <Persona: Laura Rodriguez>]>
```

Descendente

```
>>> >>> Persona.objects.order_by('-nombre')
<QuerySet [<Persona: Laura Rodriguez>, <Persona: Jose Perez>, <Persona: Antonio Suarez>]>
>>>
```

Acceso a datos (QuerySets/ORM)

Acceso a Registros Específicos:

```
>>> Persona.objects.all()[0]
<Persona: Jose Perez>
>>> Persona.objects.all()[1]
<Persona: Laura Rodriguez>
>>> Persona.objects.all()[2]
<Persona: Antonio Suarez>
>>>
```

Acceso a datos (QuerySets/ORM)

Acceso a Registros Específicos (Ascendente):

```
>>> Persona.objects.order_by('nombre')[0:1]
<QuerySet [<Persona: Antonio Suarez>]>
>>> Persona.objects.order_by('nombre')[0:2]
<QuerySet [<Persona: Antonio Suarez>, <Persona: Jose Perez>]>
>>> Persona.objects.order_by('nombre')[0:3]
<QuerySet [<Persona: Antonio Suarez>, <Persona: Jose Perez>, <Persona: Laura Rodriguez>]>
>>> Persona.objects.order_by('nombre')[0:4]
<QuerySet [<Persona: Antonio Suarez>, <Persona: Jose Perez>, <Persona: Laura Rodriguez>]>
>>>
```

Acceso a datos (QuerySets/ORM)

Acceso a Registros Específicos (Descendente):

```
>>> Persona.objects.order_by('-nombre')[0:3]
<QuerySet [<Persona: Laura Rodriguez>, <Persona: Jose Perez>, <Persona: Antonio Suarez>]>
>>> Persona.objects.order_by('-nombre')[0:2]
<QuerySet [<Persona: Laura Rodriguez>, <Persona: Jose Perez>]>
>>> Persona.objects.order_by('-nombre')[0:1]
<QuerySet [<Persona: Laura Rodriguez>]>
>>>
```

Acceso a datos (QuerySets/ORM)

Actualizar múltiples campos en una sola declaración

```
>>> p = Persona.objects.get(nombre='Jose')
>>> p
<Persona: Jose Perez>
>>> p.nombre = 'Santiago'
>>> p.apellido = 'Martinez'
>>> p.save()
>>>
```

Acceso a datos (QuerySets/ORM)

Actualizar múltiples campos en una sola declaración

```
>>>
Persona.objects.all().update(apellido='Linares')
3
>>> Persona.objects.all()
<QuerySet [<Persona: Santiago Linares>,
<Persona: Laura Linares>, <Persona: Antonio
Linares>]>
>>>
```

Acceso a datos (QuerySets/ORM)

Borrar objetos

```
>>> p =
Persona.objects.get(nombre='Santiago')
>>> p.delete()
(1, {'biblioteca.Persona': 1})
>>> Persona.objects.all()
<QuerySet [<Persona: Laura Linares>,
<Persona: Antonio Linares>]>
>>>
```

Acceso a datos (QuerySets/ORM)

Borrar objetos (Masivamente)

```
>>> p = Persona.objects.get(nombre='Santiago')
>>> p.delete()
(1, {'biblioteca.Persona': 1})
>>> Persona.objects.all()
<QuerySet [<Persona: Laura Linares>, <Persona: Antonio Linares>]>
>>> Persona.objects.filter(apellido='Linares').delete()
(2, {'biblioteca.Persona': 2})
>>> Persona.objects.all().delete()
(0, {'biblioteca.Persona': 0})
>>> Persona.objects.all()
<QuerySet []>
>>>
```

Acceso a datos (QuerySets/ORM)

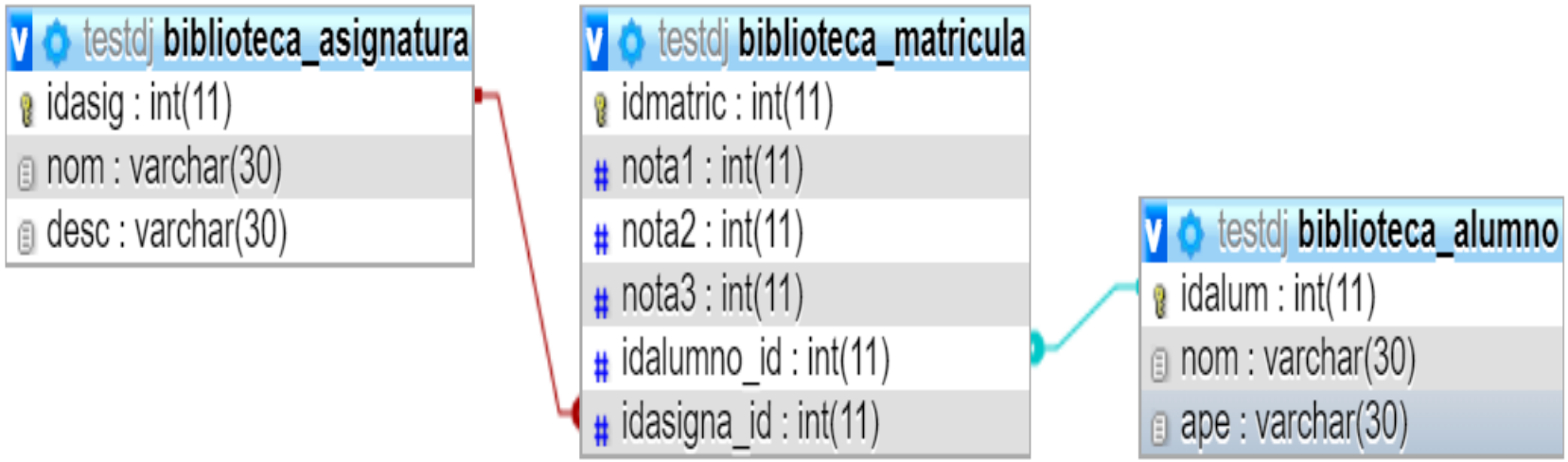
Modelo de Datos a Desarrollar:

Asumiremos los siguientes conceptos, campos y relaciones:

- Un Alumno tiene un Id, nombre y apellido.
- Una Asignatura tiene un Id , nombre y descripción.
- Una Matricula, tiene un Id, un Id de Alumno, un Id de Asignatura y 3 calificaciones del período académico.
- Un Alumno, puede cursar varias asignaturas (relación de 1 a muchos). Una Asignatura, puede ser cursada por muchos Alumnos (relación de 1 a muchos). Para romper esta relación de muchos a muchos, se tendrá una tabla intermedia, que corresponde a la matricula, donde 1 alumno, tendrá 1 registro de calificaciones por cada asignatura.

Acceso a datos (QuerySets/ORM)

Modelo de Datos a Desarrollar:



Acceso a datos (QuerySets/ORM)

Modificar Models.py e incorporar el siguiente contenido:

(Esta primera parte ya se encontraba definida)

```
from django.db import models


class Persona(models.Model):
    nombre = models.CharField(max_length=30)
    apellido = models.CharField(max_length=30)

    def __str__(self): # __unicode__ en Python 2
        return '%s %s' % (self.nombre, self.apellido)
```

Acceso a datos (QuerySets/ORM)

Modificar Models.py e incorporar el siguiente contenido:

(Lo nuevo)

```
class Alumno(models.Model):
    idalum = models.AutoField(primary_key=True)
    nom = models.CharField(max_length=30)
    ape = models.CharField(max_length=30)

    def __str__(self): # __unicode__ en Python 2
        return '%s %s %s ' % (self.idalum, self.nom, self.ape)
```

Acceso a datos (QuerySets/ORM)

Modificar Models.py e incorporar el siguiente contenido:

(Lo nuevo)

```
class Asignatura(models.Model):
    idasig = models.AutoField(primary_key=True)
    nom = models.CharField(max_length=30)
    desc = models.CharField(max_length=30)

    def __str__(self): # __unicode__ en Python 2
        return '%s %s %s' % (self.idasig, self.nom, self.desc)
```

Acceso a datos (QuerySets/ORM)

Modificar Models.py e incorporar el siguiente contenido:

(Lo nuevo)

```
class Matricula(models.Model):
    idmatric = models.AutoField(primary_key=True)
    idalumno = models.ForeignKey(Alumno, on_delete=models.CASCADE)
    idasigna = models.ForeignKey(Asignatura, on_delete=models.CASCADE)
    nota1 = models.IntegerField()
    nota2 = models.IntegerField()
    nota3 = models.IntegerField()

    def __str__(self): # __unicode__ en Python 2
        return '%s %s %s %s %s %s' % (self.idmatric, self.idalumno, self.idasigna,
self.nota1, self.nota2, self.nota3)
```

Acceso a datos (QuerySets/ORM)

Una vez definido, proceder a:

Verificar:

(TESTDJ) C:\TESTDJ\misitio>python manage.py check biblioteca

Construir Migración:

(TESTDJ) C:\TESTDJ\misitio>python manage.py makemigrations

Probar Actualizaciones:

(TESTDJ) C:\TESTDJ\misitio>python manage.py sqlmigrate biblioteca 0001

Migrar:

(TESTDJ) C:\TESTDJ\misitio>python manage.py migrate

Acceso a datos (QuerySets/ORM)

Es el momento de probar el modelo, siguiendo el siguiente procedimiento:

- Acceder a través de manage.py al shell:

  (TESTDJ) C:\TESTDJ\misitio>python manage.py shell

- Cargar la clase Alumno e ingresar los datos de prueba:

```
>>> from biblioteca.models import Alumno
>>> a1 = Alumno(nom='VANESA', ape='SALAS')
>>> a1.save()
>>> a1 = Alumno(nom='LAURA', ape='DORTA')
>>> a1.save()
```

Acceso a datos (QuerySets/ORM)

- Cargar la clase Asignatura e ingresar los datos de prueba:

```
>>> from biblioteca.models import Asignatura
>>> a2 = Asignatura(nom='MATEMATICA',desc='CCS. BASICAS')
>>> a2.save()
>>> a2 = Asignatura(nom='FISICA',desc='CCS. BASICAS')
>>> a2.save()
>>> a2 = Asignatura(nom='QUIMICA',desc='CCS. BASICAS')
>>> a2.save()
>>> a2 = Asignatura(nom='BIOLOGIA',desc='CCS. NATURALES')
>>> a2.save()
>>> a2 = Asignatura(nom='HISTORIA',desc='ESTUDIOS SOCIALES')
>>> a2.save()
```

Acceso a datos (QuerySets/ORM)

- Cargar la clase Matricula e ingresar los datos de prueba:

```
>>> from biblioteca.models import Matricula
>>> m1 = Matricula(idalumno_id=1,idasigna_id=1,nota1=20,nota2=15,nota3=17)
>>> m1.save()
>>> m1 = Matricula(idalumno_id=1,idasigna_id=2,nota1=15,nota2=14,nota3=19)
>>> m1.save()
>>> m1 = Matricula(idalumno_id=1,idasigna_id=3,nota1=17,nota2=19,nota3=19)
>>> m1.save()
>>> m1 = Matricula(idalumno_id=1,idasigna_id=4,nota1=15,nota2=15,nota3=17)
>>> m1.save()
>>> m1 = Matricula(idalumno_id=1,idasigna_id=5,nota1=13,nota2=16,nota3=19)
>>> m1.save()
>>>
```

Acceso a datos (QuerySets/ORM)

- Listar el contenido de las entidades actualizadas:

```
>>> Alumno.objects.all()
<QuerySet [<Alumno: 1 JOSE PEREZ>, <Alumno: 2 VANESA SALAS>, <Alumno: 3 LAURA DORTA>]>
>>> Asignatura.objects.all()
<QuerySet [<Asignatura: 1 MATEMATICA CCS. BASICAS>, <Asignatura: 2 FISICA CCS. BASICAS>, <Asignatura: 3 QUIMICA CCS. BASICAS>, <Asignatura: 4 BIOLOGIA CCS. NATURALES>, <Asignatura: 5 HISTORIA ESTUDIOS SOCIALES>]>
>>> Matricula.objects.all()
<QuerySet [<Matricula: 1 1 JOSE PEREZ 1 MATEMATICA CCS. BASICAS 20 15 17>, <Matricula: 2 1 JOSE PEREZ 2 FISICA CCS. BASICAS 15 14 19>, <Matricula: 3 1 JOSE PEREZ 3 QUIMICA CCS. BASICAS 17 19 19>, <Matricula: 4 1 JOSE PEREZ 4 BIOLOGIA CCS. NATURALES 15 15 17>, <Matricula: 5 1 JOSE PEREZ 5 HISTORIA ESTUDIOS SOCIALES 13 16 19>]>
>>>
```

Formularios con acceso al Modelo de Datos

Se desarrollarán ejemplos de formularios con acceso a datos:

- Modificar el archivo /misitio/misitio/urls.py, con este 1er. Contenido:

```
from django.conf.urls import path
from misitio.views import alumno_ingreso
urlpatterns = [
        path('alumno/', alumno_ingreso),
# ...
]
```

Formularios con acceso al Modelo de Datos

- Modificar el archivo /misitio/misitio/views.py, con este 1er. Contenido:

```
from django.http import HttpResponse
from django.shortcuts import render

from biblioteca.models import Alumno

def alumno_ingreso(request):
        mensajes = []
        if request.method == 'POST':
```

Formularios con acceso al Modelo de Datos

- Continuación modificación al archivo /misitio/misitio/views.py:

```
                if not request.POST.get('vNom',''):
                        mensajes.append('Por favor ingrese el
nombre del Alumno.')
                if not request.POST.get('vApe',''):
                        mensajes.append('Por favor ingrese el
apellido del Alumno.')
                if not mensajes:
                        vNom = request.POST.get('vNom','')
                        vApe = request.POST.get('vApe','')
                        nvoAlumno = Alumno(nom=vNom,ape=vApe)
                        nvoAlumno.save()
                        mensajes.append('Alumno ingresado con
éxito.')
        return render(request, 'Alumno.html',{'mensajes':mensajes})
```

Formularios con acceso al Modelo de Datos

- Continuación modificación al archivo /misitio/misitio/views.py:

```
            if not request.POST.get('vNom',''):
                mensajes.append('Por favor ingrese el nombre del Alumno.')
            if not request.POST.get('vApe',''):
                mensajes.append('Por favor ingrese el apellido del Alumno.')
            if not mensajes:
                vNom = request.POST.get('vNom','')
                vApe = request.POST.get('vApe','')
                nvoAlumno = Alumno(nom=vNom,ape=vApe)
                nvoAlumno.save()
                mensajes.append('Alumno ingresado con éxito.')
        return render(request, 'Alumno.html',{'mensajes':mensajes})
```

Formularios con acceso al Modelo de Datos

- Crear el archivo HTML: /misitio/misitio/templates/alumno.html

```
<!DOCTYPE html>
<html>
<head>
        <meta charset="utf-8">
        <title>REGISTRO DE ALUMNOS</title>
        <style type="text/css">
                *{
                        font: bold 20px verdana, sans-serif;
                }
                div{
                        padding: 20px;
                }
        </style>
</head>
```

Formularios con acceso al Modelo de Datos

- Continuación archivo HTML: /misitio/misitio/templates/alumno.html

```
<body>
        <div id="encabezado" align="center">
                REGISTRO DE ALUMNOS
        </div>
        <div id="captura">
        <form method="POST" action="/alumno/">{% csrf_token %}
                <table align="center">
                        <tr>
                                <td>
                                        NOMBRE:
                                </td>
                                <td>
                                        <input type="text"
                                             name="vNom">
                                </td>
                        </tr>
```

Formularios con acceso al Modelo de Datos

- Continuación archivo HTML: /misitio/misitio/templates/alumno.html

```
                        <tr>
                                <td>
                                        APELLIDO:
                                </td>
                                <td>
                                        <input type="text"
                                             name="vApe">
                                </td>
                        </tr>
                        <tr>
                                <td colspan="2" align="center">
                                        <input type="submit"
value="GUARDAR">
                                        <input type="reset" value="LIMPIAR">
                                </td>
                        </tr>
                </table>
        </form>
```

Formularios con acceso al Modelo de Datos

- Continuación archivo HTML: /misitio/misitio/templates/alumno.html

```
        </div>
        <div id="resultado" align="center">
        {% if mensajes %}
        <ul>
                {% for msj in mensajes %}
                        <li>{{ msj }}</li>
                {% endfor %}
        </ul>
        {% endif %}
        </div>
</body>
</html>
```

Formularios con acceso al Modelo de Datos

Protección de falsificación de solicitud de sitio cruzado

Uso del token {% csrf_token %}

El middleware CSRF y la etiqueta de plantilla proporcionan una protección fácil de usar contra falsificaciones de solicitudes entre sitios . Este tipo de ataque ocurre cuando un sitio web malicioso contiene un enlace, un botón de formulario o algún JavaScript destinado a realizar alguna acción en su sitio web, utilizando las credenciales de un usuario conectado que visita el sitio malicioso en su navegador. También se cubre un tipo de ataque relacionado, 'iniciar sesión CSRF', donde un sitio atacante engaña al navegador de un usuario para que inicie sesión en un sitio con las credenciales de otra persona.

Formularios con acceso al Modelo de Datos

Repetir el procedimiento con Asignatura

Formularios con acceso al Modelo de Datos

- Para el caso Matricula, efectuar las siguientes modificaciones: misitio/misitio/urls.py

```
from django.urls import path
from django.urls import re_path


from misitio.views import alumno_ingreso, matricula_ingreso
urlpatterns = [
    path('alumno/', alumno_ingreso),
    path('matricula/', matricula_ingreso),
# ...
]
```

Formularios con acceso al Modelo de Datos

- Para el caso Matricula, efectuar las siguientes modificaciones: misitio/misitio/templates/matricula.html

```
<!DOCTYPE html>
<html>
<head>
        <meta charset="utf-8">
        <title>Matricula</title>
        <style type="text/css">
                *{
                        font: bold 20px verdana, sans-serif;
                }
                div{
                        padding: 20px;
                }
        </style>
</head>
```

Formularios con acceso al Modelo de Datos

- Para el caso Matricula, continuación: misitio/misitio/templates/matricula.html

```
<body>
        <div id="encabezado" align="center">
                REGISTRO DE MATRICULA
        </div>
        <div id="captura">
        <form action="/matricula/" method="POST">{% csrf_token %}
                <table align="center">
                        <tr>
                                <td>ALUMNO:</td>
                                <td>
                                        <select name="vIdAlumno">
                                                {% for a in Alumnos %}
                                                        <option
value="{{a.idalum}}">{{a.nom}} {{a.ape}}</option>
                                                {% endfor %}
                                        </select>
                                </td>
                        </tr>
```

Formularios con acceso al Modelo de Datos

- Para el caso Matricula, continuación: misitio/misitio/templates/matricula.html

```
<tr>
    <td>ASIGNATURA:</td>
    <td>
        <select name="vIdAsignatura">
            {% for a1 in Asignaturas %}
                <option value="{{a1.idasig}}">{{a1.nom}}</option>
            {% endfor %}
        </select>
    </td>
</tr>
```

Formularios con acceso al Modelo de Datos

- Para el caso Matricula, continuación: misitio/misitio/templates/matricula.html

```
                                                    <tr>
                                <td>1RA. EVALUACIÓN:</td>
                                <td>
                                            <input type="number"
                                                name="vNota1"
                                                min="1"
                                                max="20"
                                                value="0">
                                </td>
                    </tr>
                    <tr>
                                <td>2dA. EVALUACIÓN:</td>
                                <td>
                                            <input type="number"
                                                name="vNota2"
                                                min="1"
                                                max="20"
                                                value="0">
                                </td>
                    </tr>
```

Formularios con acceso al Modelo de Datos

- Para el caso Matricula, continuación: misitio/misitio/templates/matricula.html

```
                <tr>
                    <td>3RA. EVALUACIÓN:</td>
                    <td>
                        <input type="number"
                            name="vNota3"
                            min="1"
                            max="20"
                            value="0">
                    </td>
                </tr>
                <tr>
                    <td colspan="2" align="center">
                        <input type="submit" value="GUARDAR">
                        <input type="reset" value="LIMPIAR">
                    </td>
                </tr>
            </table>
        </form>
```

Formularios con acceso al Modelo de Datos

- Para el caso Matricula, Views:
  misitio/misitio/views.py

```
def matricula_ingreso(request):
        Lista_Alumnos =  Alumno.objects.all()
        Lista_Asignaturas =
Asignatura.objects.all()
        return render(request,
'Matricula.html',{'Alumnos':Lista_Alumnos,\
                'Asignaturas':Lista_Asignaturas})
```

Formularios con acceso al Modelo de Datos

- Para el caso Matricula, continuación: misitio/misitio/views.py

```
        if request.method == 'POST':
                vIdAlumno = request.POST.get('vIdAlumno','')
                vIdAsignatura = request.POST.get('vIdAsignatura','')
                vNota1 = request.POST.get('vNota1','')
                vNota2 = request.POST.get('vNota2','')
                vNota3 = request.POST.get('vNota3','')
                nvaMatricula =
Matricula(idalumno_id=vIdAlumno,idasigna_id=vIdAsignatura,\

nota1=vNota1,nota2=vNota2,nota3=vNota3)
                nvaMatricula.save()
                return render(request,
'Matricula.html',{'mensajes':"Las calificaciones \
                        se registraron con éxito."})
```

FIN