GUIA MVT

Objetivo

Desarrollar el patrón Model-View-Template de django para exponer una vista que use plantillas y que cargue información de la base de datos.

Enunciado

Modelo

Esta guía parte del código desarrollado en la sesión previa que se encuentra explicada en "Guía configuración Django". En la sesión anterior se trabajó con la urls, la vista y la plantilla. Es hora de completar el MVC incorporando a nuestros servidores los modelos.

1. Empezamos definiendo un **modelo**:
    a. Un modelo es una representación de la estructura de los datos. Define cómo se almacenan y manejan los datos en la base de datos.
    b. Los modelos en Django se definen utilizando clases que heredan de ***django.db.models.Model.***
    c. Los modelos definen campos que representan los atributos de los datos y métodos que especifican cómo interactuar con ellos.

    Por tanto, nos ubicamos en el fichero auctions/models.py y escribimos lo siguiente:

```python
from django.db import models

class Auction(models.Model):
    title = models.CharField(max_length=150)
    description = models.TextField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
    discount_percentage = models.DecimalField(max_digits=5, decimal_places=2)
    rating = models.DecimalField(max_digits=3, decimal_places=2)
    stock = models.IntegerField()
    brand = models.CharField(max_length=100)
    category = models.CharField(max_length=50)
    thumbnail = models.URLField()

    def __str__(self):
        return self.title
```

    De esta forma estamos representado al objeto Subasta que será el protagonista de nuestro sitio web. Cada atributo se define indicando su tipo, ya que django trabaja únicamente con bases de datos relacionales (y son tipadas). Podéis encontrar los tipos disponibles en la siguiente página.

2. Ahora que se ha definido el modelo procedemos a configurar la base de datos. Como se ha comentado anteriormente, django solo trabaja con bases de datos relaciones. Por defecto, al crear un proyecto en django se incluye un fichero llamado db.sqlite en la carpeta principal. Este fichero actúa como base de datos para el aplicativo y es la configuración por defecto. **Sqlite NO NOS SERVIRÁ PARA DESPLEGARLA**, pero sí para trabajar en nuestro ordenador.

La configuración de la base de datos se establece en myFirstServer/settings.py. En concreto, en el apartado de base de datos (variable **DATABASES**).

```
settings.py
myFirstServer > settings.py > ...
73
74  # Database
75  # https://docs.djangoproject.com/en/5.0/ref/settings/#databases
76
77  DATABASES = {
78      'default': {
79          'ENGINE': 'django.db.backends.sqlite3',
80          'NAME': BASE_DIR / 'db.sqlite3',
81      }
82  }
```

Como vamos a utilizar la configuración por defecto, no realizamos ninguna modificación en este punto. Sin embargo, es necesario verificar que ya declaramos la nueva aplicación (*auctions*) en la variable **`INSTALLED_APPS.`** Este paso se hizo en la sesión anterior.

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

De esta forma Django se encargará de gestionar con la base de datos los modelos que definamos en la aplicación **auctions/models.py.**

3. Procedemos a comprobar cómo actúa el ORM de Django. El acrónimo ORM significa "Object-Relational Mapping" (Mapeo Objeto-Relacional). Es un enfoque de programación que permite a los desarrolladores interactuar con bases de datos relacionales utilizando objetos de un lenguaje de programación orientado a objetos, como Python, en lugar de escribir consultas SQL directamente.

   El propósito principal del ORM es abstraer la complejidad de las operaciones de la base de datos, permitiendo a los desarrolladores manipular datos de una manera más natural y orientada a objetos. Esto significa que puedes trabajar con los datos de la base de datos utilizando clases y métodos (POO) de un lenguaje de programación, en lugar de preocuparte por la sintaxis específica del SQL.

   En resumen, un ORM facilita la interacción con bases de datos relacionales al mapear las tablas y relaciones de la base de datos a objetos y relaciones en un lenguaje de programación orientado a objetos, lo que simplifica el desarrollo de aplicaciones y mejora la legibilidad del código.

   Para delegar en django las operaciones de los modelos ejecutaremos, **tras haber activado el entorno virtual de la sesión anterior (djangoServer),** los siguientes comandos:

   - `py manage.py makemigrations auctions`

- Este comando se encarga de detectar los cambios que ha habido en los modelos de la aplicación shops. Si ha habido cambios, se genera un resumen de las operaciones (escritos en Python) que se harán en la base de datos (en nuestro caso el de sqlite) y se guardarán en auctions/migrations/.
- **IMPORTANTE**: crea las operaciones, pero no las ejecuta. Eso lo hará el tercer comando (**migrate**).
- Al ser la primera vez, el resultado de este comando debería ser parecido a:

```
Migrations for auctions:
  auctions/migrations/0001_initial.py
    - Create model Auction
```

- `py manage.py sqlmigrate auctions 0001`
  - Este comando es **opcional** y sirve para ver cómo son las operaciones en lenguaje de base de datos que se han creado en el paso anterior. Dicho de otra forma. Con ***makemigrations*** se crea el resumen de los cambios y con ***sqlmigrate*** se puede consultar las *queries* en las que se traducirá.

```
BEGIN;
--
-- Create model Auction
--
CREATE TABLE "auctions_auction" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "title" varchar(150) NOT NULL, "description" text NOT NULL, "price" decimal NOT NULL, "discount_percentage" decimal NOT NULL, "rating" decimal NOT NULL, "stock" integer NOT NULL, "brand" varchar(100) NOT NULL, "category" varchar(50) NOT NULL, "thumbnail" varchar(200) NOT NULL, "images" text NOT NULL CHECK ((JSON_VALID("images") OR "images" IS NULL)));
COMMIT;
```

- `py manage.py migrate`
  - Este comando es el encargado de aplicar todas las migraciones de todos los aplicativos del proyecto.

```
Operations to perform:
  Apply all migrations: auctions
Running migrations:
  Applying auctions.0001_initial... OK
```

4. Para comprobar que se ha realizado correctamente las operaciones, podemos acceder a la base de datos. Hay varias formas de acceder al db.sqlite.
   a. Instalarte un cliente de sqlite. Puedes encontrar un instalador en esta página.
   b. Acceder con la API Shell que ofrece Django.
   c. Acceder a través de la página de administración de Django.

   Profundizaremos en las dos últimas.

5. **Acceso por API Shell.** Basta con escribir el siguiente comando en el terminal:

- `py manage.py shell`

De esta forma accedemos a una consola de Python que permite hacer consultas sobre el modelo. Para realizar la inserción de un conjunto de productos que usar de ejemplo en la base de datos podéis utilizar el "Script bd.txt" que se os facilita junto a este guion.

IMPORTATE incluir la importación del modelo: "from auctions.models import Auction", que se os facilita en la primera línea.

En el script se os facilita 4 *queries* básicas para ir familiarizándoos con el Shell.

Ejercicio 1

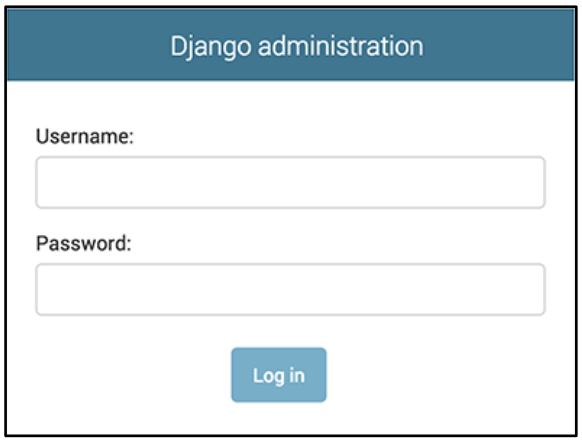
¿Por qué solo nos devuelve el título de las subastas? ¿Cómo podemos añadir el precio?

6. **Acceso a través de la página de administración.** Django ofrece por defecto una página de administración que te permite gestionar a través de una página web (visualmente más agradable que por consola) los elementos registrados en base de datos.

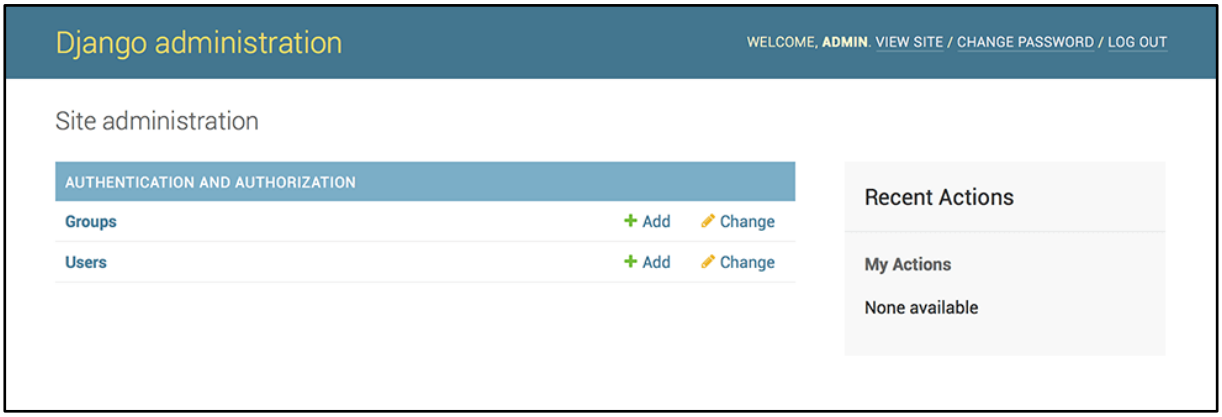
   Para configurarla, únicamente la primera vez, es necesario ejecutar los siguientes comandos:

   - `py manage.py createsuperuser`.
     - Este comando creará un usuario maestro para garantizar el acceso de administrador. Solicitará nombre de usuario, email y escribir dos veces una contraseña. Estas credenciales (usuario y contraseña) será la que necesitaréis para loggearos. **¡Apuntadla!**

   - `py manage.py runserver`
     - Este comando ya lo hemos lanzado varias veces. Levanta el servidor. Y ahora accederemos a la página "http://127.0.0.1:8000/admin/".



     - Usarás las credenciales generadas con el comando anterior.
     - Una vez loggeado correctamente, podrás consultar los roles de los usuarios registrados:



Pero ahora queremos poder administrar los modelos de la aplicación **auctions**; es decir, las 10 subastas que hemos dado de alta mediante el api Shell.

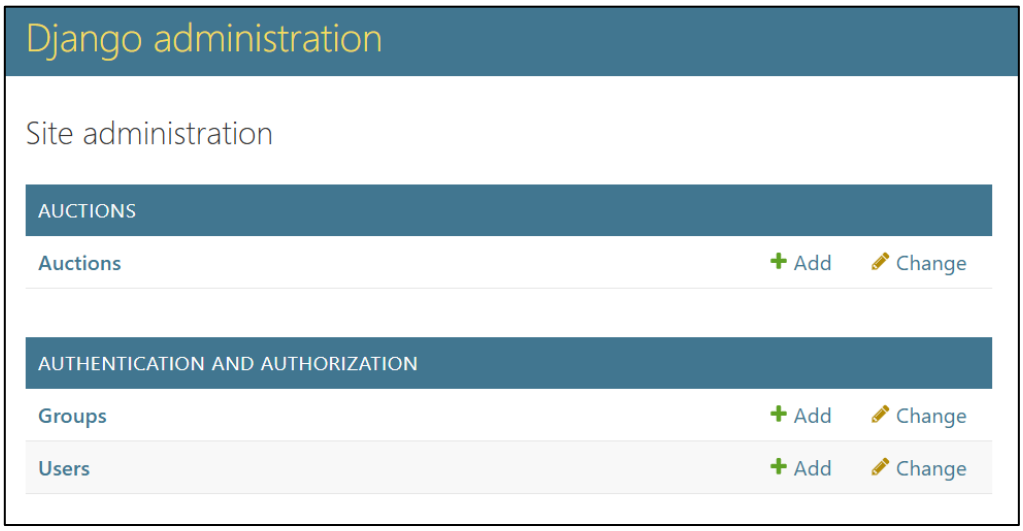
Para indicar que queremos que la aplicación Auction sea mostrada por el api de administración, hay que ir a auctions/admin.py y añadir el siguiente código:

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

admin.site.register(Auction)
```

Con el código anterior, declaramos Auction. Si tuviésemos más modelos, habría que ir añadiéndolos también. De esta forma se controla cuáles si y cuáles no se exponen.

Si volvemos a la página "http://127.0.0.1:8000/admin/", ahora ya veremos la opción de Auctions (tanto de la aplicación como del modelo).



Y desde aquí podemos añadir un nuevo producto, y consultar, modificar y eliminar los ya existentes (operaciones CRUD).

Ejercicio 2

Usando la página de administración, realiza las siguientes acciones:

- Crea una nueva subasta. El objeto subastado será un reloj. Los campos quedan a vuestra elección.
- Edita la subasta del Iphone 9. Cambia el título para que sea una Iphone 12 y que valga 1350€. El descuento debe ser 0.
- Borra la subasta referente al Huawei Matebook X Pro.

Vista

7. Las **vistas** son componentes que manejan la lógica de la aplicación y procesan las solicitudes del usuario. Las vistas en Django pueden ser funciones o clases, y son responsables de interactuar con los modelos para recuperar, crear, actualizar o eliminar datos, y luego devolver una respuesta adecuada al usuario.

8. Ampliamos el funcionamiento del auctions/views.py para que ahora devuelva dos nuevas vistas: la información del detalle de una subasta y la lista de subastas.

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

from .models import Auction

def index(request):
    auctions = Auction.objects.all()
    template = loader.get_template("auctions/index.html")
    context = { "auction_list": auctions }
    return HttpResponse(template.render(context, request))

def detail(request, auction_id):
    auction = Auction.objects.get(id=auction_id)
    context = { "auction": auction }
    return render(request, "auctions/detail.html", context)
```

Los dos métodos (index y detail) ofrecen las dos formas de devolver una plantilla. Ambas son completamente válidas. No obstante, se suele implementar la segunda para reducir el número de líneas de código.

El funcionamiento de ambos métodos es el siguiente:

- Se carga de la base de datos una serie de subastas.
- Se define el contexto, que es la variable que permite asociar variables a las plantillas. De esta forma, podemos acceder desde la plantilla al modelo (en este ejemplo son subastas).
- Se carga la plantilla asociada.

9. También es necesario realizar cambios en auctions/urls.py:

```
from django.urls import path
from .views import hello_view, index, detail

urlpatterns = [
    path('hello/', hello_view, name='hello'),
    path("", index, name="index"),
    path("<int:auction_id>", detail, name="detail"),
]
```

Así declaramos una nueva vista; la del detalle del producto. Esta nueva vista utilizará un identificador del producto.

Ahora nos fijamos en el atributo "name" que está definiendo un path. Se relacionará con la forma de redirigir entre plantillas. Esto nos permite mantener independientes la ruta de los servicios.

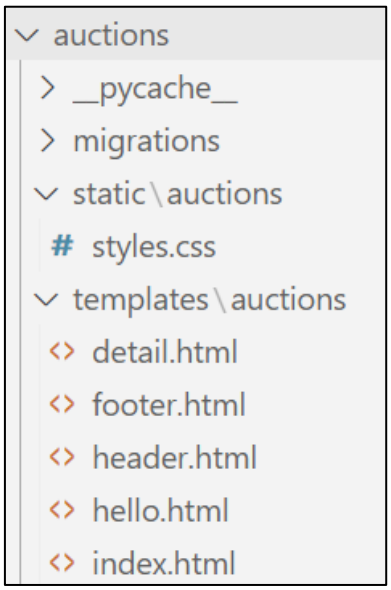
Plantillas

10. Las **plantillas** son archivos HTML que contienen marcadores de plantilla especiales para insertar datos dinámicos y lógica de presentación. Las plantillas permiten separar la lógica de presentación del resto de la aplicación y son responsables de renderizar la interfaz de usuario que se envía al navegador del usuario.

11. Creamos la carpeta "static/auctions" en "auctions" y añadimos el fichero style.css que se proporciona con este guion.

12. En la carpeta "templates/auctions" en "auctions" añadimos el html de las 2 nuevas plantillas.

    El resultado final de los dos últimos pasos (11 y 12) debe ser:



13. El **index.html** sigue la misma estructura que hemos visto en ejercicios de HTLM-CSS-Javascript. Pero vamos a analizar los cambios que se han implementado:

```
{% load static %}

<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Lista de subastas</title>
  <link rel="stylesheet" type="text/css" href="{% static 'auctions/styles.css' %}">
</head>
```

- Se añade **{% load static %}** que es una etiqueta de plantilla en Django que se utiliza para cargar el contenido estático, como archivos CSS, JavaScript e imágenes, en una plantilla HTML. Cuando Django renderiza una plantilla que contiene esta etiqueta, se asegura de que el contenido estático definido en la configuración de tu proyecto se pueda acceder correctamente.
- Se incluye el fichero de CSS y se utiliza **static** para referenciar la dirección del fichero.

```
<body>
  <header>
    <h1>Subastas Web</h1>
    <nav>
      <ul>
        <li><a href="{% url 'index' %}">Listado</a></li>
      </ul>
    </nav>
  </header>
```

- Las referencias entre plantillas se basan el uso de url. Url es una etiqueta de plantilla en Django que se utiliza para generar URLs de manera dinámica en las plantillas HTML.

  En este caso específico, 'index' es el nombre de la vista (en el campo “name) para la cual queremos generar la URL. Cuando utilizas {% url 'index' %}, Django buscará una ruta definida en el archivo urls.py de tu proyecto que coincida con el nombre 'index' y generará la URL correspondiente.

```
<div class="container">
  <h2>Nuestras subastas</h2>
  <div>
    {% if auction_list %}
        {% for auction in auction_list %}
      <div class="item-details">
      <img src="{{auction.thumbnail}}"" alt="Thumbnail">
      <div class="info">
      <h2>{{auction.title}}</h2>
      <p>{{auction.description}}<p>
      <p><strong>Precio:</strong><span>{{auction.price}}</span>€</p>
      <p><strong>Stock:</strong><span>{{auction.stock}}</span></p>
      </div>
    </div>
        {% endfor %}
    {% else %}
    <p>No hay subastas disponibles.</p>
    {% endif %}
  </div>
</div>
```

- Las etiquetas de plantilla, como **{% if %}, {% for %}, {% block %},** etc., se utilizan para agregar lógica de control de flujo a la plantilla.

  En este caso estamos comprobando si se ha definido en el contexto de la plantilla una variable llamada auction_list. En caso de que así sea, se recorrerá la lista para pintar los diferentes productos. Y si no es así, se mostrará el mensaje de que no hay subastas disponibles.

- Los marcadores de plantilla se encierran entre llaves dobles, como {{ variable }}, y se utilizan para mostrar datos dinámicos en la plantilla.

```
{% include './footer.html' %}
```

- Existe herencia e inclusión entre plantillas. De esta forma, podemos reutilizar HTML entre diferentes plantillas. La plantilla **detail.html** utiliza esta herencia.

14. Arranca el servidor y accede a las nuevas vistas creadas.
    a. `python manage.py runserver`
    b. Accede a http://127.0.0.1:8000/auctions/
    c. Accede a http://127.0.0.1:8000/auctions/1

15. Finalmente, vamos a desarrollar la funcionalidad para dar de alta una nueva subasta. Por tanto, añade la plantilla **"new.html"** facilitada.

16. Añade en urls.py:
    path('new/', create, name='create'),

17. Modifica views.py para **incluir** la función "create". No debes perder las funciones "index" y "detail". Por tanto, añade el import al principio del fichero mientras que la función "créate" debe ir al final del fichero.

```
from .forms import AuctionForm

def create(request):
    if request.method == "POST":
        form = AuctionForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('index')
    else:
        form = AuctionForm()

    return render(request, 'auctions/new.html', {'form': form})
```

18. Como veis, nos hace falta un nuevo fichero llamado "forms.py". Los formularios en Django se utilizan por varias razones:
    a. **Validación automática de datos**: Garantiza que los datos del usuario sean correctos antes de ser guardados.
    b. **Manejo de errores:** Muestra mensajes claros si los datos no son válidos.
    c. **Seguridad**: Protege contra ataques CSRF (Cross-Site Request Forgery).
    d. **Facilidad de integración con modelos:** Los ModelForm simplifican la creación de formularios basados en los modelos de la base de datos.
    e. **Prevención de inyecciones SQL:** Django maneja el escape de datos automáticamente, evitando vulnerabilidades.
    f. **Reutilización:** Los formularios son fáciles de reutilizar en diferentes vistas y aplicaciones.

Damos de alta el formulario de la siguiente manera:

```
from django import forms
from .models import Auction

class AuctionForm(forms.ModelForm):
    class Meta:
        model = Auction
        fields = [
            'title', 'description', 'price', 'discount_percentage',
            'rating', 'stock', 'brand', 'category', 'thumbnail'
        ]
```

19. Accede a http://127.0.0.1:8000/auctions/new/

Ejercicio 3

Crea una nueva subasta usando la plantilla. Puedes usar los valores que quieras.

Ejercicio 4

Implementa la funcionalidad para poder editar una subasta. La plantilla se debe llamar “edit.html”, la vista se debe llamar “edit” y la urls “/edit”.

Ejercicio 5

Modifica la página web para incluir las siguientes mejoras:

- La plantilla “header.html” incluya redirección a la página de creación de nueva subasta.
- Desde el listado de subastas haya dos botones nuevos:
  - “Ver subasta” que redirige al detalle de la subasta.
  - “Editar subasta” que sitúa en la página de edición creada en el ejercicio 4.

Ejercicio 6

Desarrolla un botón que permita eliminar una subasta. Debe incluir un botón en el listado, igual que se ha realizado en el ejercicio anterior para ver y editar.