taller demostrativo, en el que los participantes podrán ejecutar el código suministrado y reconocer los diferentes aspectos del funcionamiento, organización y metodología para el desarrollo de un pequeño proyecto web basado en Django

TALLER DE INICIACION AL DESARROLLO WEB CON PYTHON.

Framework Django

Autores

Ing. Marco León Mora Méndez,
Ing. Sandra Milena Cruz,
Ing. Mauricio Mahecha,
Ing. Adriana Rincón

CONTENIDO

TABLA DE FIGURAS

INTRODUCCIÓN

Este taller de Iniciación al Desarrollo Web con Python (Django), se desarrolla en el marco de Expo-Industria 2022 del Centro de Industria y Construcción, a desarrollarse en un total de nueve horas, durante tres días.

Dada la corta duración del evento y la amplitud de la temática a tratar, será un taller demostrativo, en el que los participantes podrán ejecutar el código suministrado y reconocer los diferentes aspectos del funcionamiento, organización y metodología para el desarrollo de un pequeño proyecto web basado en Django.

OBJETIVOS DEL TALLER

El objetivo de estas jornadas es crear una pequeña aplicación Web para presentar los productos de un supermercado, clasificados por categorías, con las siguientes funcionalidades:

- Listar las categorías de productos
- Listar los productos por categoría
- Presentar las características de un producto
- Acumular productos en el carrito de compra
- Administrar el carrito de compras para eliminar productos o cambiar cantidades
- Ejecutar la compra (se enviará un correo con la factura)
- Un usuario con permisos especiales (superusuario) tendrá acceso a la consola de administración de los datos (Dashboard)

CARACTERÍSTICAS DE DJANGO

Django es un framework de desarrollo web de código abierto, escrito en Python, que respeta el patrón de diseño modelo–vista–controlador (MVC). Fue desarrollado originalmente para gestionar páginas web 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.

La meta fundamental de Django es facilitar la creación de sitios web complejos. Django pone énfasis en el re-uso, la conectividad y extensibilidad de componentes, el desarrollo rápido y el principio «DRY» (del inglés Don't Repeat Yourself, «No te repitas»). El lenguaje Python es usado en todos los componentes del framework, incluso en configuraciones, archivos y en sus modelos de datos.

Django viene con autenticación incorporada a través de sesiones y el uso de tokens.

Algunas de las características de Django son:

- Seguridad: Django tiene activados mecanismos incluidos para proteger la base de datos, formularios y JavaScript.
- Escalabilidad: se puede utilizar el framework para un desarrollo sencillo, hasta uno mucho más complejo, ambos casos funcionarán de manera estable y con rapidez.
- Interfaz: Su interfaz para acceso a la base de datos y hacer consultas es sumamente buena.
- Portable: Al estar escrito en Python, se puede ejecutar en muchas plataformas como Windows, OS X, entre otras, dándole muchísima libertad al programador al momento de ejecutar las aplicaciones.
- Programar con Django será muy fácil si está familiarizado con Python ya que al desarrollar un sitio web será más ágil por la ausencia de tipos de datos en las variables, el manejo de objetos y la sintaxis sencilla (similar al pseudocódigo) que permite realizar tareas complejas sin la necesidad de escribir tantas líneas de código, siendo más práctico que otros lenguajes de programación como PHP.
- A comparación de otros frameworks, Django está prácticamente listo para empezar un proyecto y terminarlo muy rápidamente.
- Proporciona otras características como módulos de autenticación, serialización y deserialización JSON, API routing, etc.

SABERES PREVIOS Y HERRAMIENTAS A UTILIZAR

Se espera que el participante tenga experiencia en maquetado de páginas web, utilizando HTML, CSS y JavaScript y en el lenguaje Python, igualmente se recomienda conocer Bootstrap y JQuery.

El participante debe comprender o al menos tener alguna familiaridad con los siguientes conceptos:

- Arquitectura Cliente- Servidor
- Protocolo HTTP
- Programación Orientada a Objetos

Para el presente taller se requiere que los computadores a utilizar tengan instalado Python (con Pip), Git, Visual Studio Code; también se requiere conectividad a Internet.

ARQUITECTURA DJANGO

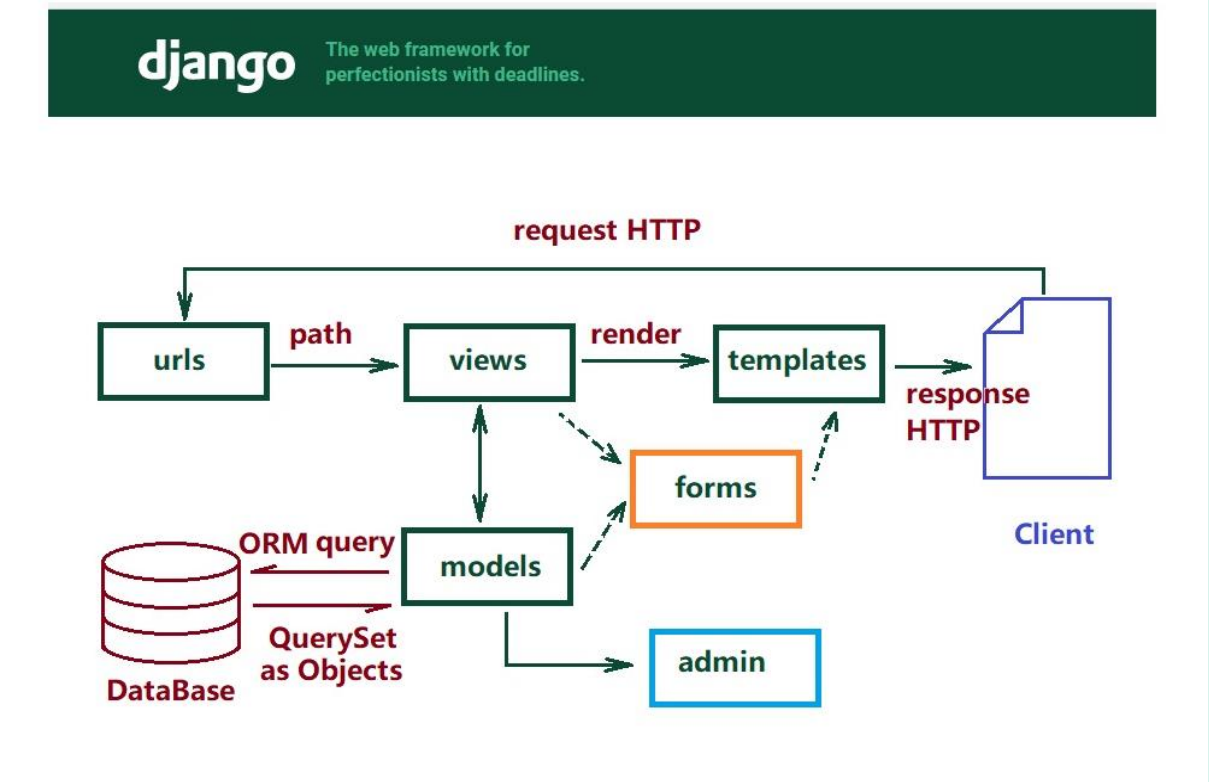


Figura 1. Arquitectura Django

La organización de los componentes de un proyecto Django se basa en una variante de MVC (Modelo – Vista – Controlador), denominada MTV (Model – Template – View Controller), como se observa en la figura 1).

Las peticiones HTTP desde el cliente (Al hacer clic en un enlace o escribir una dirección), se accede al mapa de URLs, en este archivo cada ruta está asociado con una view (una función o una clase), si se necesita algún dato se solicitará este a model, el cual a su vez generara la consulta a la base de datos, cuando los datos han sido traídos estos son enviados al template que contiene la lógica de presentación para estos. Luego de "pintar" (render) la página esta se enviará al navegar que hizo la solicitud.

Adicionalmente, Django tiene incluido un mecanismo que facilita la creación de formularios (forms) para ingreso de datos, que se pueden asociar a las tablas del modelo.

Es posible, configurar el modelo para ser controlado en un dashboard para la visualización y administración de los datos (CRUD).

ORGANIZACIÓN DE ARCHIVOS EN UN PROYECTO DJANGO

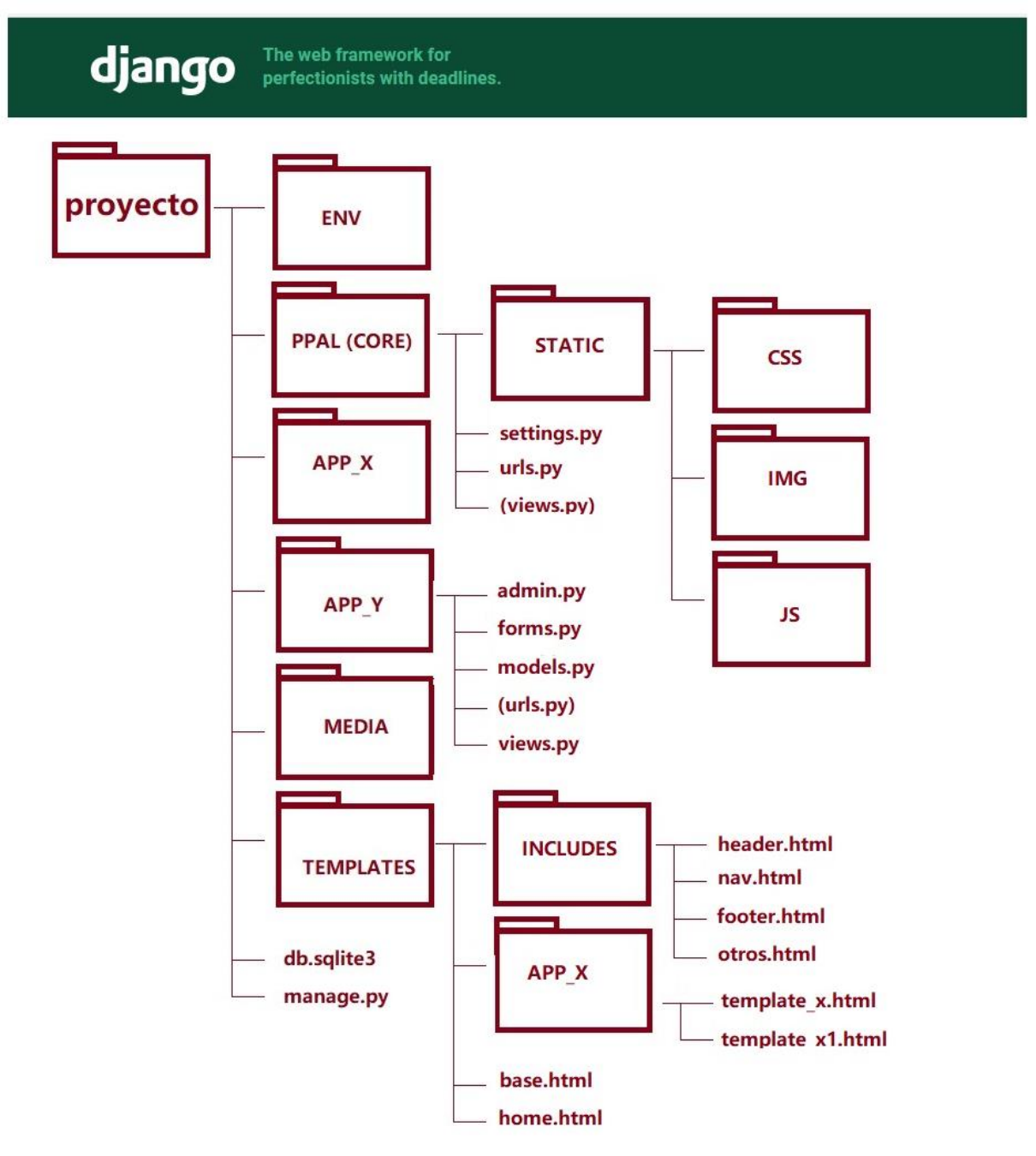


Figura 2. Estructura de carpetas y archivos del proyecto

Cuando se desarrolla un proyecto Django utilizando ambiente virtual, es recomendado seguir el siguiente esquema de organización de carpetas y archivos (ver Fig. 2)

1. Todo el proyecto se encontrará dentro de una carpeta con el nombre del proyecto.
2. El ambiente virtual creará una carpeta, normalmente denominada 'env' con contenido ya predefinido. Se recomienda no modificar este contenido.

3. Al crear el proyecto django, se creará una carpeta núcleo del proyecto (core). Esta carpeta podría tener el mismo nombre de la carpeta general, u otro.
4. En ella se generán algunos archivos Python, de los cuales serán modificados por el desarrollador los siguientes:
   4.1. 'Settings.py', donde se registran las variables globales del sistema.
   4.2. 'urls.py', donde se definen las rutas de navegación (paths).
   4.3. Opcionalmente se puede crear un archivo 'views.py' (no está por defecto) para insertar las funciones del controlador asociadas a las rutas, aunque podrían usarse los archivos 'views.py' de los módulos (denominados Apps) a crear posteriormente.
5. Normalmente se crea dentro de la carpeta core (3.) otra carpeta 'static' que contendrán todos los archivos adicionales necesarios para el maquetado Html, organizados dentro de subcarpetas, ahí también deben estar las imágenes necesarias para el maquetado (logos, etc.). En caso de proyectos complejos, cada App puede contener su propia carpeta 'static', las que serán recogidas en una sola cuando se haga el despliegue del proyecto (deployment) en un servidor remoto.
6. Cuando se crea un módulo funcional (App), este residirá en su propia carpeta (app_x, app_y, etc, en la figura 2.), se debe registrar su existencia en el archivo 'settings.py'. En ella están, entre otros, los siguientes archivos, a modificar por el programador:
   6.1. Un archivo 'views.py' que contendrá la lógica del controlador en forma de funciones o de clases.
   6.2. Un archivo 'models.py' para definir el modelo de datos, cada tabla se representa por una clase.
   6.3. Un archivo 'admin.py' para registrar los modelos que se quieran manipular dentro de la consola de administración (dashboard CRUD)
   6.4. Opcionalmente se puede crear un archivo 'urls.py', donde se definen las rutas de navegación (paths) dentro de la app. En este caso se debe referenciar este archivo dentro de las rutas del archivo 'urls.py' del proyecto (3.)
   6.5. Opcionalmente se puede crear un archivo 'forms.py', para crear formularios, aprovechando la funcionalidad de django que permite asociar el modelo al formulario y presenta herramientas como la generación de código html y la validación de datos del formulario.
7. Las plantillas html se organizan dentro de una carpeta 'templates' a crear por el programador. Esta carpeta puede existir dentro de cada App o crear una sola a nivel de la carpeta del proyecto. En cualquier caso, se debe registrar su existencia en el archivo 'settings.py' (1.). Se recomienda organizar las plantillas dentro de subcarpetas.
   7.1. Una carpeta 'includes' donde estarán los bloques de código html específicos, a utilizar en todos o la mayoría de las plantillas, por ejemplo 'header.html', 'nav.html', 'footer.html', etc.
   7.2. Usualmente se crean los archivos 'base.html' y 'home.html' (la página inicial) en la carpeta 'templates' del Core.
8. Cuando el proyecto implique la manipulación de archivos de documentos o de imágenes (propios de la funcionalidad, no de la presentación), se creará una carpeta 'media', en la que, para mejorar la organización, se pueden crear subcarpetas por temas.

CREACIÓN DEL ENTORNO VIRTUAL Y DEL PROYECTO (CORE)

INICIAR PROYECTO DJANGO

Pasos para instalar Django, entorno virtual e iniciar proyecto base:

1. Instalar python
2. Instalar git
3. Crear carpeta con nombre del proyecto
4. En la carpeta, botón derecho para desplegar el menú contextual y clic sobre '**git bash here**'. Se abre consola git.
5. Crear enviroment: *python -m venv env* [1] (env: nombre del ambiente virtual)

```
user@tipo1 MINGW64 ~/Documents/GitHub/workshopDjango (master)
$ python -m venv env
```

6. Activar el virtual environment: *source env/Scripts/activate*

```
user@tipo1 MINGW64 ~/Documents/GitHub/workshopDjango (master)
$ source env/Scripts/activate
(env)
```

7. instalar Django en el enviroment: *pip install django*

```
user@tipo1 MINGW64 ~/Documents/GitHub/workshopDjango (master)
$ pip install django
Collecting django
  Downloading Django-4.1.2-py3-none-any.whl (8.1 MB)
Collecting tzdata
  Downloading tzdata-2022.4-py2.py3-none-any.whl (336 kB)
Collecting asgiref<4,>=3.5.2
  Using cached asgiref-3.5.2-py3-none-any.whl (22 kB)
Collecting sqlparse>=0.2.2
  Using cached sqlparse-0.4.3-py3-none-any.whl (42 kB)
Installing collected packages: tzdata, sqlparse, asgiref, django
Successfully installed asgiref-3.5.2 django-4.1.2 sqlparse-0.4.3 tzdata-2
```

8. verificar la instalación: *pip freeze* Se desplegará la lista de productos instalados con su versión

[1] En adelante, los comandos a introducir en la consola Git bash tendrán este formato de texto.

```
user@tipol MINGW64 ~/Documents/GitHub/workshopDjango (master)
$ pip freeze
asgiref==3.5.2
Django==4.1.2
sqlparse==0.4.3
tzdata==2022.4
(env)
```

9. crear proyecto Django: *django-admin startproject nombreProyecto .* (colocar el punto separado, para generar en la carpeta raíz)

```
user@tipol MINGW64 ~/Documents/GitHub/workshopDjango (master)
$ django-admin startproject workShop .
(env)
```

INICIAR SERVIDOR Y NAVEGADOR

Levantar el servidor y navegar a la página de presentación:

1. Levantar el servidor: *python manage.py runserver*

```
user@tipol MINGW64 ~/Documents/GitHub/workshopDjango (master)
$ python manage.py runserver
Watching for file changes with StatReloader
[04/Oct/2022 15:55:45] "GET / HTTP/1.1" 200 10681
```

2. probar con navegador en *127.0.0.1:8000* o *localhost:8000*. Se debe desplegar una página como la siguiente:

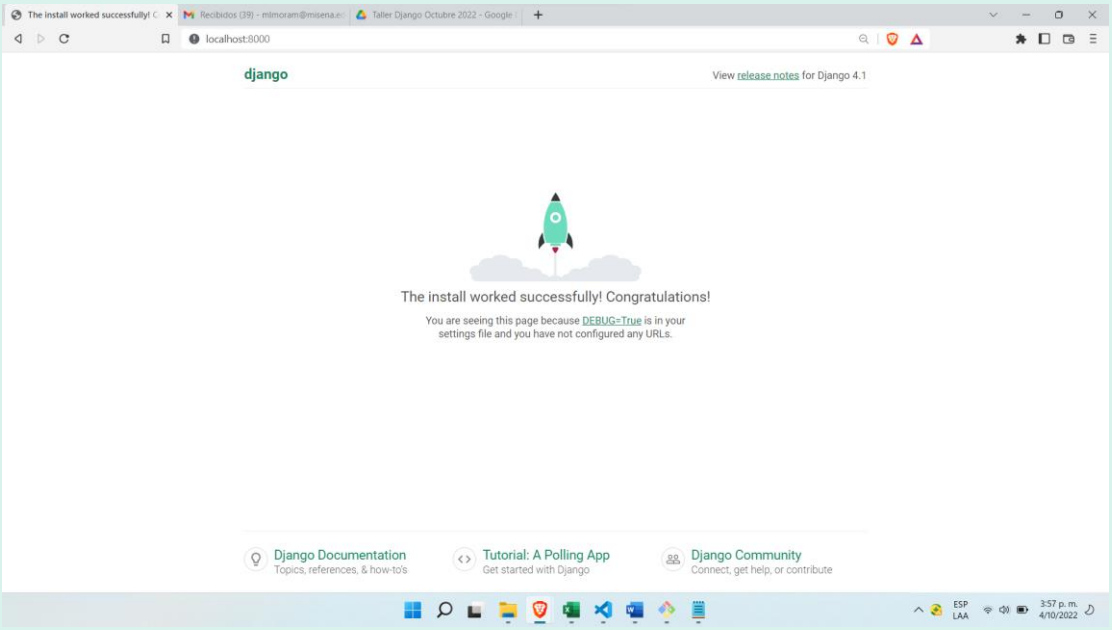


Figura 3. Página de inicio de un proyecto Django

USANDO ARCHIVOS ESTÁTICOS

DEFINIR LA PÁGINA INICIAL (HOME)

Todo el código html hace parte de las plantillas, debe crear una carpeta ‘templates’ a nivel de la carpeta exterior y registrarla en ‘settings.py’, en la variable ‘TEMPLATES’, ‘DIRS’: [‘templates’].

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': ['templates'],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

Dentro de la carpeta ‘templates’ crear un archivo ‘home.html’ con el código HTML según su diseño de la página de inicio:

```
<!DOCTYPE html>
<html lang="es">
<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>WorkShop Django</title>
</head>
<body>
    <h1>Bienvenido al WorkShop Django del Centro de Industria</h1>
</body>
</html>
```

Ahora, hay que crear la ruta (ver Figura 1). En el archivo ‘urls.py’:

```python
from django.contrib import admin
from django.urls import path
from . import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.home, name='home'),
]
```

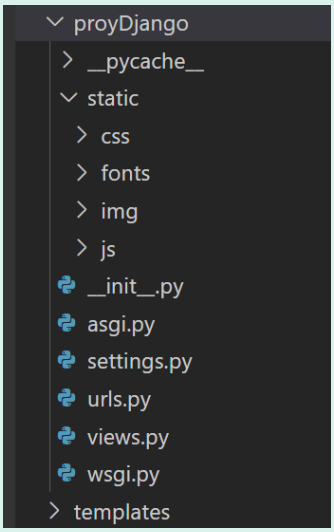
En la figura 1, el siguiente bloque es VIEWS. Crear un archivo 'views.py' en la misma carpeta de 'urls.py' (la carpeta 'proyDjango'), con el siguiente código:

```python
from django.shortcuts import render

def home(request):
    return render(request, 'home.html')
```

En la consola levantar el servidor (*python manage.py runserver*), y verificar los mensajes por si hay errores, si no, ir al navegador y refrescar la página, debe visualizarse el contenido de 'home.html'.

INCLUDES - CSS-JS-IMG-FONTS

Los archivos adicionales necesarios para el maquetado de la página se organizan en carpetas que deben estar registradas en el archivo de configuración ('settings.py').



En la carpeta del proyecto principal (proyDjango) crear una carpeta 'static'

Dentro de ella las carpetas necesarias según el tipo de archivos a utilizar, por ejemplo: 'css', 'img', 'fonts', 'js', etc. En ellas incluir los archivos a utilizar.

Además, en el archivo 'settings.py' incluir las siguientes variables:

```python
STATIC_URL = 'static/'
STATIC_ROOT = BASE_DIR /'static'
STATICFILES_DIRS = [
    'proyDjango/static',
]
```

Ahora, hay que indicar, dentro del código html, las rutas a esos archivos, utilizando comandos Django.

Dentro de la plantilla 'home.html', incluir las siguientes líneas:

Al inicio del código indicando que debe cargar archivos desde la carpeta 'static'.

```
{% load static %}
<!DOCTYPE html>
<html lang="es">
```

Para cada enlace a un archivo estático, incluir, su ruta, a partir de la carpeta 'static'.

```
    <link rel="stylesheet" href="{% static 'css/bootstrap.min.css' %}">
    <link rel="stylesheet" href="{% static 'css/estilo.css' %}">
    <script src="{% static 'js/bootstrap.min.js' %}"></script>
```

De manera similar para otros enlaces a imágenes, fonts, etc.

Observar que, para incluir código Django dentro de un template, este debe estar encerrado entre los caracteres: `{% aquí el código Django %}`

FACTORIZANDO COMPONENTES DE PLANTILLAS

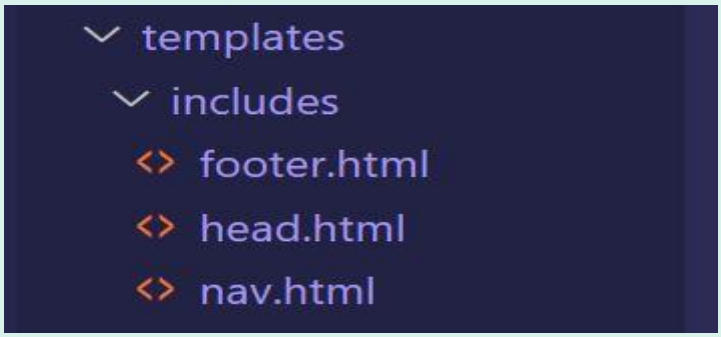
La factorización de plantilla consiste en separar en archivos '.html' los bloques de etiquetas que pueden ser reutilizados, además, crear una plantilla que sirva de base para insertar las diferentes vistas según sea necesario.

1. Crear archivo 'base.html', dentro de la carpeta 'templates' (ver ejemplo en el anexo 1. Observar que:
    a. utiliza el comando `{% load static %}` ya que existen referencias a archivos estáticos.
    b. Cuando se desee incluir un archivo html con alguna funcionalidad, se usan includes, ejemplo: `{% include 'includes/head.html' %}`. Hay que definir la ruta a partir de la carpeta 'templates'.
    c. En esta aplantilla se define el sitio donde se desee agregar el contenido concreto de una o varias plantillas:

```
                    {% block content %}

                    {% endblock %}
```

HEADER, NAV, FOOTER

En una carpeta ‘includes’ se crea un archivo html por cada segmento de código a reutilizar, según su funcionalidad, Cree los archivos ‘head.html’, ‘nav.html’, ‘footer.html’.



ARCHIVO ‘HEAD.HTML’

Observe que ya contiene algunos comandos Django para desplegar el nombre del usuario, que se explicarán en el desarrollo del taller.

```
{% load static %}
<div class="row mb-2" id="encabezado">
    <div class="col-8" id="titulo">
        <h1>SUPERMARKET</h1>
        <div class="col-12" id="nombreUsuario">
            {% if user.id is None %}
                <h6>visitante</h6>
            {% else %}
                <h6>{{ user.first_name }} - {{ user.rol }}</h6>
            {% endif %}
        </div>
    </div>
    <div class="col-4">
        <img class="d-block mx-auto" src="{% static 'img/supermarket.png' %}" alt="">
    </div>
</div>
```

ARCHIVO ‘NAV.HTML’

```
<nav class="navbar navbar-expand-lg bg-light">
    <div class="container-fluid">
        <a class="navbar-brand" href="#">Menu:</a>
        <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-
target="#navbarSupportedContent"
            aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle
navigation">
            <svg class="navbar-toggler-icon" xmlns="http://www.w3.org/2000/svg" width="16" height="16"
                fill="var(--color-text1)" viewBox="0 0 16 16">
                <path fill-rule="evenodd"
                    d="M2.5 12a.5.5 0 0 1 .5-.5h10a.5.5 0 0 1 0 1H3a.5.5 0 0 1-.5-.5zm0-4a.5.5 0 0 1
.5-.5h10a.5.5 0 0 1 0 1H3a.5.5 0 0 1-.5-.5zm0-4a.5.5 0 0 1 .5-.5h10a.5.5 0 0 1 0 1H3a.5.5 0 0 1-.5-
.5z" />
            </svg>

        </button>
        <div class="collapse navbar-collapse" id="navbarSupportedContent">
            <ul class="navbar-nav me-auto mb-2 mb-lg-0">
                <li class="nav-item">
                    <a class="nav-link active" aria-current="page" href="">
                        <span>
                            <svg xmlns="http://www.w3.org/2000/svg" class="icon icon-tabler icon-
tabler-home" width="30"
                                height="30" viewBox="0 0 24 24" stroke-width="2" stroke="currentColor"
fill="none"
                                stroke-linecap="round" stroke-linejoin="round">
                                <path stroke="none" d="M0 0h24v24H0z" fill="none"></path>
                                <polyline points="5 12 3 12 12 3 21 12 19 12"></polyline>
                                <path d="M5 12v7a2 2 0 0 0 2 2h10a2 2 0 0 0 2 -2v-7"></path>
                                <path d="M9 21v-6a2 2 0 0 1 2 -2h2a2 2 0 0 1 2 2v6"></path>
                            </svg>
                        </span>
                        Inicio
                    </a>
                </li>
                {% if user.id is not None %}

                <li class="nav-item">
                    <a class="nav-link" href="#">
                        <span>
                            <svg xmlns="http://www.w3.org/2000/svg" class="icon icon-tabler icon-
tabler-square-asterisk"
```

```
                                width="30" height="30" viewBox="0 0 24 24" stroke-width="2"
stroke="currentColor"
                                fill="none" stroke-linecap="round" stroke-linejoin="round">
                                <path stroke="none" d="M0 0h24v24H0z" fill="none"></path>
                                <rect x="4" y="4" width="16" height="16" rx="2"></rect>
                                <path d="M12 8.5v7"></path>
                                <path d="M9 10l6 4"></path>
                                <path d="M9 14l6 -4"></path>
                            </svg>
                        </span>
                        Cambiar contraseña
                    </a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="#">
                        <svg xmlns="http://www.w3.org/2000/svg" class="icon icon-tabler icon-tabler-
shopping-cart"
                            width="30" height="30" viewBox="0 0 24 24" stroke-width="2"
stroke="currentColor"
                            fill="none" stroke-linecap="round" stroke-linejoin="round">
                            <path stroke="none" d="M0 0h24v24H0z" fill="none"></path>
                            <circle cx="6" cy="19" r="2"></circle>
                            <circle cx="17" cy="19" r="2"></circle>
                            <path d="M17 17h-11v-14h-2"></path>
                            <path d="M6 5l14 1l-1 7h-13"></path>
                        </svg>
                        Carrito
                    </a>
                </li>

                <li class="nav-item dropdown">
                    <a class="nav-link dropdown-toggle" href="#" role="button" data-bs-
toggle="dropdown"
                        aria-expanded="false">

                        <span>
                            <svg xmlns="http://www.w3.org/2000/svg"
                                class="icon icon-tabler icon-tabler-layout-bottombar-collapse"
width="30" height="30"
                                viewBox="0 0 24 24" stroke-width="2" stroke="currentColor" fill="none"
                                stroke-linecap="round" stroke-linejoin="round">
                                <path stroke="none" d="M0 0h24v24H0z" fill="none"></path>
                                <path
                                    d="M20 6v12a2 2 0 0 1 -2 2h-12a2 2 0 0 1 -2 -2v-12a2 2 0 0 1 2 -
2h12a2 2 0 0 1 2 2z">
```

```
                            </path>
                            <path d="M20 15h-16"></path>
                            <path d="M14 8l-2 2l-2 -2"></path>
                        </svg>
                    </span>
                    Opciones
                </a>
                <ul class="dropdown-menu">
                    <li><a class="dropdown-item" href="#">Categorias</a></li>
                    <li><a class="dropdown-item" href="#">Productos por Categoria</a></li>
                    <li><a class="dropdown-item" href="#">Todos los productos</a></li>
                </ul>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="#">
                    <span>
                        <svg xmlns="http://www.w3.org/2000/svg" class="icon icon-tabler icon-
tabler-logout"
                            width="30" height="30" viewBox="0 0 24 24" stroke-width="2"
stroke="currentColor"
                            fill="none" stroke-linecap="round" stroke-linejoin="round">
                            <path stroke="none" d="M0 0h24v24H0z" fill="none"></path>
                            <path
                                d="M14 8v-2a2 2 0 0 0 0 -2 -2h-7a2 2 0 0 0 0 -2 2v12a2 2 0 0 0 0 2 2h7a2
2 0 0 0 2 -2v-2">
                            </path>
                            <path d="M7 12h14l-3 -3m0 6l3 -3"></path>
                        </svg>
                    </span>
                    Salir
                </a>
            </li>
            {% else %}
            <li class="nav-item">
                <a class="nav-link" href="">
                    <span>
                        <svg xmlns="http://www.w3.org/2000/svg" class="icon icon-tabler icon-
tabler-user-plus"
                            width="30" height="30" viewBox="0 0 24 24" stroke-width="2"
stroke="currentColor"
                            fill="none" stroke-linecap="round" stroke-linejoin="round">
                            <path stroke="none" d="M0 0h24v24H0z" fill="none"></path>
                            <circle cx="9" cy="7" r="4"></circle>
                            <path d="M3 21v-2a4 4 0 0 1 4 -4h4a4 4 0 0 1 4 4v2"></path>
                            <path d="M16 11h6m-3 -3v6"></path>
```

```
                            </svg>
                        </span>
                        Registrarse
                    </a>
                </li>

                <li class="nav-item">
                    <a class="nav-link" href="#">
                        <span>
                            <svg xmlns="http://www.w3.org/2000/svg" class="icon icon-tabler icon-
tabler-login"
                                width="30" height="30" viewBox="0 0 24 24" stroke-width="2"
stroke="currentColor"
                                fill="none" stroke-linecap="round" stroke-linejoin="round">
                                <path stroke="none" d="M0 0h24v24H0z" fill="none"></path>
                                <path
                                    d="M14 8v-2a2 2 0 0 0 -2 -2h-7a2 2 0 0 0 -2 2v12a2 2 0 0 0 2 2h7a2
2 0 0 0 2 -2v-2">
                                </path>
                                <path d="M20 12h-13l3 -3m0 6l-3 -3"></path>
                            </svg>
                        </span>
                        Ingresar
                    </a>
                </li>
                {% endif %}
            </ul>
        </div>
    </div>
</nav>
```

ARCHIVO ‘FOOTER.HTML’

```
<footer class="my-5 pt-5 text-muted text-center text-small">
    <p class="mb-1" id="tituloPie">© WorkShop Django del Centro de Industria</p>
    <ul class="list-inline">
        <li class="list-inline-item"><a href="#">Contacto</a></li>
        <li class="list-inline-item"><a href="#">Terminos de uso</a></li>
        <li class="list-inline-item"><a href="#">Soporte</a></li>
    </ul>
</footer>
```

LAS APPS DE DJANGO

Django pide que se organice el código por módulos funcionales llamados 'apps'. En este taller se creará una app para manejar los productos y otra para los usuarios.

CREAR APP DE USUARIOS Y DE PRODUCTOS

En la consola, desactivar (bajar) el servidor local, con la combinación de teclas '<CTRL> + C'. La consola debe continuar con el ambiente virtual activo. Digitar los siguientes comandos, para crear las apps:

```
(env)
mlmor@MarcoLeon MINGW64 ~/OneDrive/Documents/SENA/ADSI 2022/TallerDjango/wo
$ python manage.py startapp appUsuarios
```

```
(env)
mlmor@MarcoLeon MINGW64 ~/OneDrive/Documents/SENA/ADSI 2022/TallerDjango/wo
$ python manage.py startapp appProductos
```

Se generarán las carpetas con los nombres asignados y por dentro una serie de carpetas y archivos que iremos reconociendo posteriormente.

A continuación, debe registrar estas apps recién creadas. En 'settings.py', agregar:

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

CREAR MODELOS (USUARIOS, CATEGORÍA, PRODUCTO, CARRO)

Nuestro ejercicio consiste en ofrecer una serie de productos organizados por categorías, además del control de usuarios, incluyendo los pedidos en proceso o ya realizados; así que se debe implementar

en Django el diseño de la BD, de la siguiente figura (no se muestran las PK, Django las crea directamente):

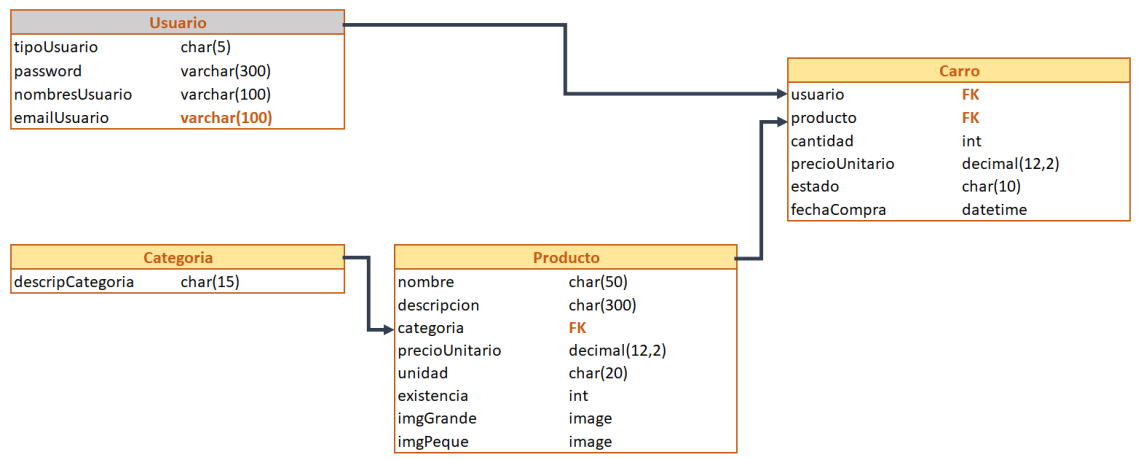


Figura 4. Modelo de la Base de Datos

CREAR EL MODELO USUARIO PARA ADMINISTRAR LOS USUARIOS, EL LOGÍN Y EL LOGOUT.

Para crear el primer modelo, dentro de la appUsuarios, en el archivo 'models.py', ingresar el siguiente código:

```
from django.db import models
from django.contrib.auth.models import AbstractBaseUser, BaseUserManager
# Create your models here.
class MyAccountManager(BaseUserManager):
    def create_user(self, first_name, last_name, email, username, password= None):
        if not email:
            raise ValueError('El usuario debe tener un correo')

        if not username:
            raise ValueError('El usuario debe tener un username')

        user = self.model(
            email = self.normalize_email(email),
            username = username,
            first_name = first_name,
            last_name = last_name,
        )

        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_superuser(self, first_name, last_name, email, username, password):
```

```
        user = self.create_user(
            email = self.normalize_email(email),
            username = username,
            password = password,
            first_name = first_name,
            last_name = last_name,
        )
        user.is_admin= True
        user.is_active = True
        user.is_staff = True
        user.is_superadmin = True
        user.save(using=self._db)
        return user

class Usuario(AbstractBaseUser):
    ROLES= (
        ('admin', 'admin'),
        ('cliente', 'cliente'),
    )

    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    username = models.CharField(max_length=50, unique=True)
    email = models.CharField(max_length=100, unique=True)
    rol = models.CharField(max_length=30, choices=ROLES, default='cliente')

    #Atributos de DJANGO
    date_joined = models.DateTimeField(auto_now_add=True)
    last_login = models.DateTimeField(auto_now_add=True)
    is_admin = models.BooleanField(default=False)
    is_staff = models.BooleanField(default=False)
    is_superadmin = models.BooleanField(default=False)
    is_active = models.BooleanField(default=False)

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['username', 'first_name', 'last_name']

    objects = MyAccountManager()

    def __str__(self):
        return '{} {}'.format(self.first_name, self.last_name)

    def has_perm(self, perm, obj=None):
        return self.is_admin
```

```
    def has_module_perms(self, add_label):
        return True

    class Meta:
        verbose_name_plural = "Usuarios"
```

Django maneja la definición de las tablas como clases (Programación Orientada a Objetos) y cada registro de una tabla (una fila) es considerado un objeto o instancia de la clase.

Las columnas de la tabla son definidas como atributos de clase. Observe que se ha creado una superclase (`MyAccountManager`) y clases heredadas (`AbstractBaseUser, BaseUserManager`) que utiliza una gran cantidad de funcionalidades predefinidas por Django para el manejo de usuarios.

Django utiliza por defecto una clase (tabla) 'users' para el manejo de usuarios, pero nosotros la hemos sobrescrito en la clase 'Usuario', por lo que es necesario registrarlo en 'settings.py'. Agregar la variable de entorno AUTH_USER_MODEL :

```
WSGI_APPLICATION = 'proyDjango.wsgi.application'
AUTH_USER_MODEL = 'appUsuarios.Usuario'
```

De la misma manera, dentro de la appProductos, en el archivo 'models.py', ingresar el siguiente código:

```
from django.db import models

# Create your models here.
class Categoria(models.Model):
    descripCategoria = models.CharField(max_length=100, null=False)

    def __str__(self):
        return self.descripCategoria

    class Meta:
        verbose_name = 'categorias'
        verbose_name_plural = 'categorias de productos'


class Producto(models.Model):
    nombre = models.CharField(max_length=100, null=False)
    descripcion = models.CharField(max_length=300, null=True)
    precioUnitario = models.DecimalField(max_digits=8, decimal_places=2)
    unidad = models.CharField(max_length=10, null=False)
```

```
    existencia= models.IntegerField(null=True)
    imgGrande = models.ImageField(upload_to='productos', null=True)
    imgPeque = models.ImageField(upload_to='iconos', null=True)
    categoria= models.ForeignKey(Categoria,  on_delete=models.CASCADE, null=False)

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

En las clases anteriores, se han incluido algunos métodos:

1. El método '__str__()' es utilizado para definir que atributos (columnas) mostrar cuando se esté desplegando un objeto (fila).
2. La definición 'class META' se utiliza para cambiar nombres en la consola de administración.

MIGRAR MODELOS

Si usted usa imágenes administradas desde el modelo, debe instalar la librerio Pillow y agregar un par de variables de estado en 'settings.py'. Por favor siga los pasos del anexo 3.

Para que Django pueda crear las tablas en la BD (en este caso SQLite), se deben ejecutar los siguientes comandos en la consola (makemigrations y migrate)

```
(env)
mlmor@MarcoLeon MINGW64 ~/OneDrive/Documents/SENA/ADSI 2022/TallerDjango
rkShop
$ python manage.py makemigrations
Migrations for 'appProductos':
  appProductos\migrations\0001_initial.py
    - Create model Categoria
    - Create model Producto
```

```
(env)
mlmor@MarcoLeon MINGW64 ~/OneDrive/Documents/SENA/ADSI 2022/TallerDjango/wo
rkShop
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, appProductos, 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 appProductos.0001_initial... 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 auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
```

Si usted usa un aplicativo para administrar directamente la BD, en este caso SQLiteStudio, se puede observar que Django ha creado, además de las tablas de nuestro modelo de BD, algunas otras utilizadas para propósitos administrativos.

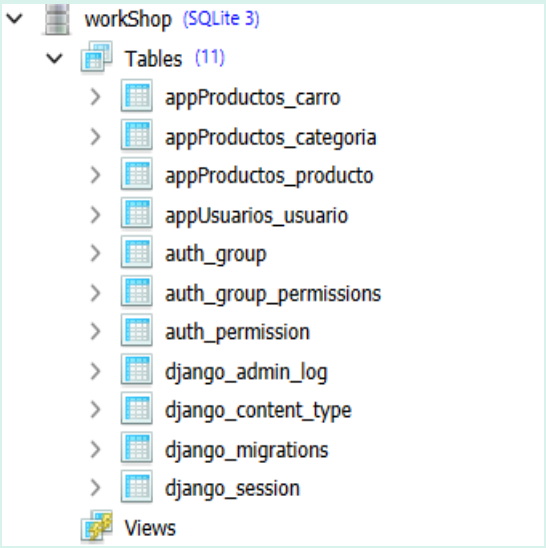


Figura 5. Tablas creadas por Django al migrar

DASHBOARD DE ADMINISTRACIÓN

La interfaz de administración de Django es una gran solución para la administración de la BD por parte de un usuario con permisos especiales (super-usuario). Sin ninguna configuración adicional, obtiene un área de administración potente, totalmente personalizable y protegida por inicio de sesión que muestra todos los datos de la aplicación.

CREAR SUPER USUARIO

Para desplegar y poder operar la consola de administración, ejecutar el siguiente comando para crear un superusuario por consola:

```
(env)
mlmor@MarcoLeon MINGW64 ~/OneDrive/Documents/SENA/ADSI 2022/TallerDjango/work
p
$ winpty python manage.py createsuperuser
Email: mlmoram@misena.edu.co
Username: marcoleon
First name: Marco
Last name: Mora
Password:
Password (again):
Superuser created successfully.
```

Se recomienda con insistencia, guardar en algún archivo los datos del superusuario, para su uso posterior.

Una vez realizados los pasos anteriores, en un navegador escriba la url 'localhost:8000/admin' y se presentará la siguiente ventana. Ingrese sus credenciales para realizar el login.

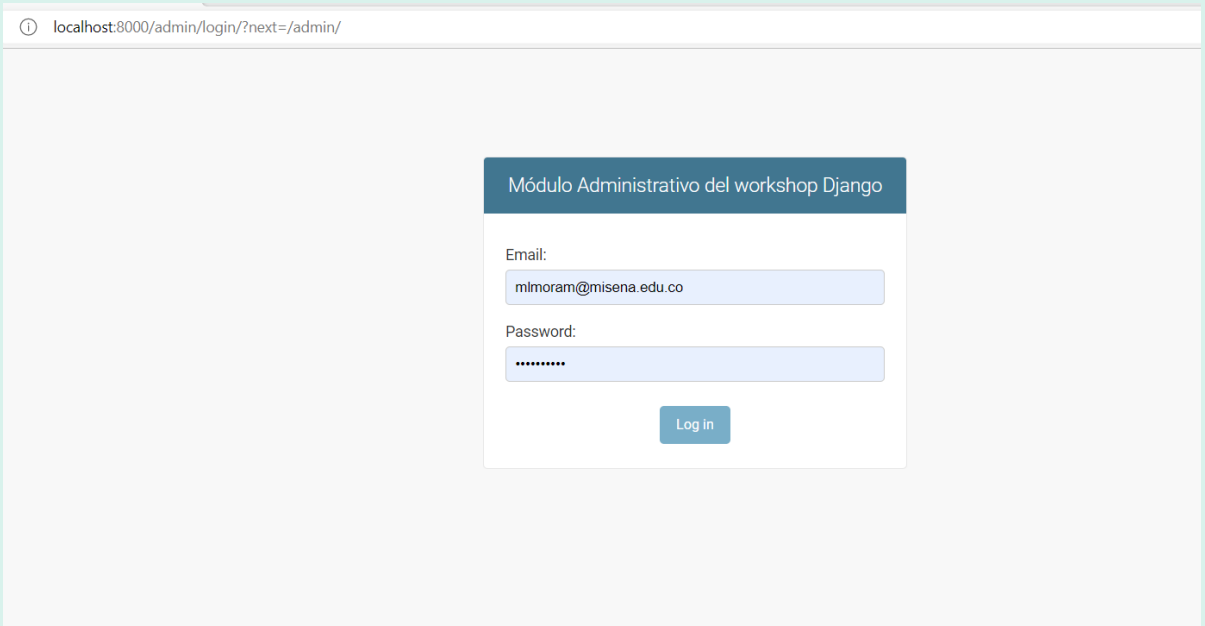


Figura 6. Login en el módulo de administración

Aparecerá la siguiente consola de administración, sin ninguna de las tablas creadas:

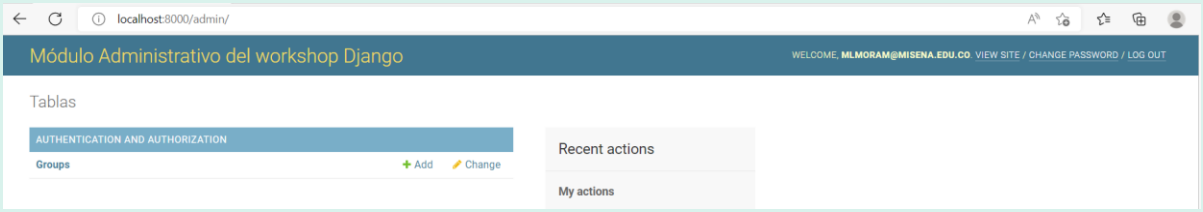


Figura 7. Módulo administrativo sin tablas registradas

REGISTRAR MODELOS EN ADMIN

Es necesario registrar las tablas y configurarlas en los archivos 'admin.py' en cada app, para poder visualizarlas en el dashboard.

En el archivo 'admin.py' de la appUsuarios:

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

# Register your models here.
admin.site.register(Usuario)
```

En el archivo 'admin.py' de la appProductos:

```
from django.contrib import admin
from .models import *
# Register your models here.
#-------------------------------------------
admin.site.register(Categoria, CategoriaAdmin)
admin.site.register(Producto, ProductoAdmin)
admin.site.register(Carro, CarroAdmin)
```

PERSONALIZAR DASHBOARD

Modificaciones a 'admin.py' de la appProductos para cambiar la presentación del dashboard:

```python
from django.contrib import admin
from .models import *
# Register your models here.
#-----------------------------------------
class CategoriaAdmin(admin.ModelAdmin):
    list_display = ['id','descripCategoria']

admin.site.register(Categoria, CategoriaAdmin)

#-----------------------------------------
class ProductoAdmin(admin.ModelAdmin):
    list_display = ['nombre','descripcion', 'existencia']

admin.site.register(Producto, ProductoAdmin)

#-----------------------------------------
class CarroAdmin(admin.ModelAdmin):
    list_display = ['usuario','producto', 'cantidad', 'estado']

admin.site.register(Carro, CarroAdmin)
```

la consola cambia a a:

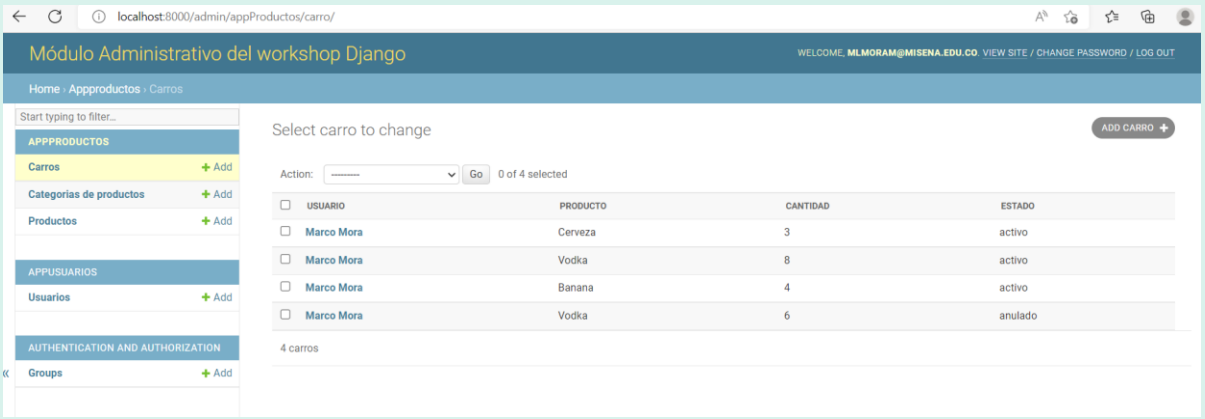


Figura 8. Módulo administrativo con tablas registradas

En este momento se pueden crear algunos registros de categorías usando este módulo de administración. Ver en el anexo 2 la lista de categorías a utilizar.

Por ahora, no agregar productos, ya que se asociarán imágenes y para el efecto hay que realizar alguna configuración que se verá más adelante.

CREANDO RUTAS

Para hacer operativo el sitio, se deben implementar los ciclos de flujo del proceso, según lo requiere Django. Para el lograrlo hay que tener muy presente la figura 1.

CREAR RUTA PARA VER CATEGORÍAS

Vamos a mostrar primero las categorías de productos:

1. ACTIVAR URL EN EL ENLACE(HTML)

En la plantilla del menú (‘includes/nav.html’) insertar la ruta en el atributo ‘href’ de la etiqueta ancla <a>, correspondiente. Se encuentra dentro de las opciones del submenú desplegable:

```
        <ul class="dropdown-menu">
        <li><a class="dropdown-item" href="{% url 'categorias' %}">Categorias</a></li>
```

Se agrega un comando Django para construir una url y el nombre que se le asigna a esa ruta. Este nombre debe ser el mismo con el siguiente paso.

2. CREAR PATH(URLS.PY)

En ‘urls.py’ del proyecto principal, insertar la siguiente ruta, que inccluira el contenido de las rutas internas de la app:

```
from django.contrib import admin
from django.urls import path, include
from . import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.home, name='home'),
    path('productos/', include('appProductos.urls')),
]
```

Crear el archivo ‘urls.py’ dentro de appProductos e insertar el siguiente código:

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

urlpatterns = [
    path('categorias/', views.verCategorias, name='categorias'),
]
```

La ruta anterior tiene tres atributos: la url asignada (‘categorias/’), la función asociada en la vista y el nombre que le hemos dado en la plantilla del menu.

3. CREAR VISTA(VIEWS.PY)

En el archivo ‘views.py’ de la appProductos se debe Implementar la lógica necesaria, en Python, para consular en la BD las categorías a mostrar, ingrese el siguiente código:

```
from django.shortcuts import render
from .models import *

# Create your views here.
def verCategorias(request):
    #Consultar categorias
    listaCateg = Categoria.objects.all()

    #ensamblar context
    context = {
        'categorias': listaCateg,
        'titulo': 'Categorias de Productos del Supermercado',
    }
    #renderizar
    return render(request, 'productos/categorias.html', context)
```

Observaciones al código anterior:

1. La función trae como argumento el objeto ‘request’ propio del protocolo HTTP que se envia del cliente al servidor.
2. Se realiza una consulta a la BD, a la tabla Categoria para recuperar todos los registros. Observe Django utiliza un ORM[2] propio para ello y traduce internamente al lenguaje SQL del DBMS utilizado (en este caso SQLite)
3. Para renderizar datos en el template, estos se pasan en un diccionario, en este caso llamado ‘context’.

4. RENDERIZANDO EL HTML

Por último, hay que crear la plantilla para visualizar la lista de categorías. Para mantener ordenado todo, la plantilla estará en una subcarpeta ‘productos’ dentro de ‘templates’.

Cree el archivo ‘categorias.html’ con el siguiente código:

```
{% extends 'base.html' %}
```

[2] El mapeo objeto-relacional ORM 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.

```
{% block content %}
<br>

<div class="mb-3 descripcion text-center">
    <h4>{{titulo}}</h4>
</div>

<div class="row">

    <table class="table">
        <thead>
            <th>Descripción Categoria</th>
            <th></th>
        </thead>
        <tbody>
            {% for cat in categorias %}
            <tr>
                <td>{{cat.descripCategoria}}</td>
                <td><a href="{% url 'productos' cat.id %}" type="button" class="btn btn-info">Listar
Productos</a></td>
            </tr>
            {% endfor %}
        </tbody>
    </table>
</div>

{% endblock %}
```

Observaciones al código anterior:

1. El comando {% extends 'base.html' %} para indicar que esta plantilla se insertara dentro de ‘base.html’
2. Los comandos {% block congtent %} y {% endblock %}, idénticos a los que existen en ‘base.html’ que le dicen a Django que lo que está por dentro debe renderizarse en el mismo sitio de base.html
3. Para renderizar datos, estos se indican dentro de ‘{{ }}’ como en ‘{{ titulo }}’. Observe que el nombre dentro de las llaves debe ser el mismo del diccionario ‘context’ enviado desde la vista.
4. Para visualizar los productos se crea una tabla con una fila de encabezados y dentro del <tbody> una fila por cada producto, por lo que se debe tener un ciclo {% for cat in categorias %} con su cierre {% endfor %}.
5. Para cada categoria (llamada por nosotros ‘cat’ dentro del ciclo, se renderizan los atributos ‘descripcategoria’ e ‘id’ (que es la Primary Key del producto en la tabla). Este forma parte de la nueva URL que permitirá enlazar la ruta para visualizar cada producto (a crear más adelante.

CREAR RUTA PARA VER PRODUCTOS POR CATEGORIA

PASOS PREVIOS

Antes de mostrar los productos, es conveniente poblar la BD incluyendo las imágenes de los productos. Use el script del anexo 2 y siga los pasos del anexo 3 para manipular las imágenes a mostrar.

Copie dentro de la carpeta 'media', las carpetas con las imágenes grandes (''productos) y pequeñas (iconos) suministradas.

Para crear la ruta de productos se siguen los mismos pasos:

1. ACTIVAR URL EN EL ENLACE(HTML)

En este caso, tener en cuenta que este paso ya se realizó: `<a href="{% url 'productos' cat.id %}",`

2. CREAR PATH(URLS.PY)

Agregar a 'urls.py' de la appProductos:

```
urlpatterns = [
    path('categorias/', views.verCategorias, name='categorias'),
    path('productos/<str:idCategoria>', views.verProductosCategoria, name='productos'),
]
```

Observaciones:

1. La ruta incluye un 'slug' con el código de la categoría, especificando que es de tipo 'string'. Este código se agregó en el paso 1 (cat.id).
2. La función asociada es 'verProductosCategoria' en la vista.
3. El nombre de la ruta debe ser el mismo del paso 1.

3. CREAR VISTA(VIEWS.PY)

Agregar el siguiente código en 'views.py' de appProductos:

```
def verProductosCategoria(request, idCategoria):
    #Consultar categorias
    idCat = int(idCategoria)
    nombreCat = Categoria.objects.get(id=idCat)
```

```
    listaProductos = Producto.objects.filter(categoria= idCat)

    #ensamblar context
    context = {
        'productos': listaProductos,
        'titulo': 'Productos de la categoria ' + str(nombreCat),
    }
    #renderizar
    return render(request, 'productos/productos.html', context)
```

Observaciones:

1. Se declara el argumento 'idCategoria' de la función.
2. Este id (la misma PK del registro en la BD) se convierte a entero (se hubiera podido hacer en la ruta.
3. Se consulta primero la tabla Categoria para recuperar el nombre de la categoría a mostrar, pero el objeto 'nombreCat' realmente contiene todas las columnas.
4. Se realiza la consulta en la tabla Producto, filtrando por la FK categoría. Este debe ser exactamente el mismo nombre definido en la clase correspondiente en 'models.py'.
5. Todos los productos de la categoría residen ahora en una lista 'listaProductos'.
6. Se ensambla el diccionario 'context' con la lista de productos y un título.
7. El titulo toma el nombre de la categoría. Observar que aquí, al concatenar strings, no se toma el objeto completo (sería un error) sino lo que devuelve el método '__str_()' de la clase (ver la definición de la clase en 'models.py').

4. RENDERIZANDO EL HTML

Cree el archivo 'productos/productos.html' en la carpeta 'templates':

```
{% extends 'base.html' %}
{% block content %}
<br>
    <div class="mb-3 descripcion text-center">
        <h2>{{titulo}}</h2>
    </div>

    <div class="row">

        <table class="table">
            <thead>
                <th></th>
                <th>nombre</th>
                <th>Descripción</th>
                <th>Precio</th>
                <th>Existencia</th>
                <th></th>
```

```
            </thead>
            <tbody>
                {% for prod in productos %}
                <tr>
                    <td><img src="{{prod.imgPeque.url}}" alt="X"> </td>
                    <td>{{prod.nombre}}</td>
                    <td>{{prod.descripcion}}</td>
                    <td>{{prod.precioUnitario}}</td>
                    <td>{{prod.existencia}}</td>
                    <td><a href="{% url 'un_producto' prod.id %}" type="button" class="btn btn-
info">Ver detalle</a></td>
                </tr>
                {% endfor %}
            </tbody>
        </table>
    </div>
{% endblock %}
```

Observaciones:

1. La etiqueta <img> enlaza a una imagen. Este tema se tratará enseguida.
2. Se crea una nueva URL para desplegar los detalles de un producto.

CREAR PÁGINA DETALLE

1. ACTIVAR URL EN EL ENLACE(HTML)

Ya se realizó: `<a href="{% url 'un_producto' pro.id %}",`

2. CREAR PATH(URLS.PY)

Agregar a 'urls.py' de la appProductos:

```
urlpatterns = [
    path('categorias/', views.verCategorias, name='categorias'),
    path('productos/<str:idCategoria>', views.verProductosCategoria, name='productos'),
    path('producto/<str:idProd>', views.verProducto, name='un_producto'),
]
```

3. CREAR VISTA(VIEWS.PY)

Agregar el siguiente código en 'views.py' de appProductos:

```python
def verProducto(request, idProd, msj = None):
    #Consultar
    idProd = int(idProd)
    regProducto = Producto.objects.get(id= idProd)
    #ensamblar context
    context = {
        'producto': regProducto,
        'titulo': 'Detalles de ' + str(regProducto.nombre),
    }

    if msj:
        context['mensaje']=  msj
    #renderizar
    return render(request, 'productos/producto.html', context)
```

Observaciones:

1. Se ha agregado el argumento ‘msj’ que es opcional, para usar más adelante. Esta entrada del diccionario context, se agrega solo si existe el argumento.

4. RENDERIZANDO EL HTML

Cree el archivo ‘productos/producto.html’ en la carpeta ‘templates’:

```html
{% extends 'base.html' %}
{% load static %}
{% block content %}
<h2>{{titulo}}</h2>
<div class="row row-cols-1 row-cols-md-2 mb-3 text-center">

    <div class="col-12 col-md-4 offset-md-4">
        <div class="card mb-4 rounded-3 shadow-sm">
            <div class="card-header py-3">
                <h4 class="my-0 fw-normal">{{producto.nombre}}</h4>
            </div>
            <img src="/media/productos/{{producto.imgGrande}}" alt="X">
            <div class="card-body">

                <h1 class="Card subtitle pricing-card-title">${{producto.precioUnitario}}</h1>
                <h4 class="text-muted fw-light">/{{producto.unidad}}</h4>
                <br>
                <ul class="list-unstyled mt-3 mb-4">
```

```
                    <p>{{producto.descripcion}}</p>
                    <p>{{producto.existencia}} en existencia</p>
                </ul>
                <a href="{% url 'agregarCarro' producto.id %}" type="button"
                    class="w-100 btn btn-lg btn-outline-primary">Agregar al
                    carrito</a>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

CONTROLANDO EL INGRESO DE USUARIOS

Se crean las rutas para el login y el logout de usuarios, el proceso de registro de usuarios se implementará posteriormente.

SECUENCIA DE IMPLEMENTACIÓN

1. ACTIVAR URL EN EL ENLACE(HTML)

En 'nav.html' (por claridad, se ha eliminado código no significativo):

```
            <li class="nav-item">
                <a class="nav-link active" aria-current="page" href="">Inicio</a>
            </li>
            {% if user.id is not None %}
            <li class="nav-item">
                <a class="nav-link" href="#">Cambiar contraseña</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="{% url 'carrito' %}"> Carrito </a>
            </li>Opciones
                <ul class="dropdown-menu">
                    <li><a class="dropdown-item" href="{% url 'categorias' %}">Categorias</a></li>
                </ul>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="{% url 'logout' %}">Salir </a>
            </li>
            {% else %}
            <li class="nav-item">
                <a class="nav-link" href="">Registrarse </a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="{% url 'login' %}">Ingresar </a>
            </li>
            {% endif %}
        </ul>
```

Observaciones:

1. El menú se presenta diferente si el usuario es visitante o ya ha realizado l login.

2. CREAR PATH(URLS.PY)

Agregar a ‘urls.py’ del proyecto el include:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.home, name='home'),
    path('productos/', include('appProductos.urls')),
    path('usuario/', include('appUsuarios.urls')),
]
```

Crear archive ‘urls.py’ en appUsuarios:

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

urlpatterns = [
    path('login/', views.login, name='login'),
    path('logout/', views.logout, name='logout'),
]
```

3. CREAR VISTA(VIEWS.PY)

Agregar el siguiente código en ‘views.py’ de appUsuarios:

```
from django.shortcuts import render, redirect
from .models import Usuario
from django.contrib import auth
from django.contrib.auth.decorators import login_required

def login(request):
    if request.method == 'POST':
        email = request.POST['email']
        password = request.POST['password']
        user = auth.authenticate(email=email, password=password)

        if user is not None:
            auth.login(request, user)
            return render(request, 'home.html')
        else:
            return render(request, 'usuarios/login.html', {'alarma': 'Correo o password no valido!'})
    else:
```

```
        return render(request, 'usuarios/login.html')


#********* DESACTIVACION DEL USUARIO ***********************************************************

@login_required(login_url='login')
def logout(request):
    auth.logout(request)
    return redirect('login')
```

Observaciones del ingreso:

1. El logín se realiza cuando se pulsa el botón ‘submit’ del formulario de ingreso, que se envía por método ‘POST’, el cual se verifica
2. Se requiere el correo y el password para ingresar
3. Después de intentar la autenticación (funcionalidades de Django) se verifica si está activo
4. Si lo está, lo lleva a la página inicial (home)
5. Si no, presenta de nuevo el formulario de ingreso

```
#********* DESACTIVACION DEL USUARIO ***********************************************************
@login_required(login_url='login')
def logout(request):
    auth.logout(request)
    return redirect('login')
```

6. El logout se realiza solo si ya a ingresado, para el efecto se agrega un decorador a la función: @login_required().

4. RENDERIZANDO EL HTML

En la carpeta ‘templates’ cree una carpeta ‘usuarios’ y en ella el archivo ‘login.html’

```
{% extends 'base.html' %}
{% block content %}
<br>
<div class="mb-3 descripcion text-center">
    <h4>Ingreso</h4>
</div>
<div class="col-md-4 offset-md-4 mb-3">
    <form enctype="multipart/form-data" action="{% url 'login' %}" method="POST">
        {% csrf_token %}
        <div class="row">
            <div class="mb-3 col-12">
                <label for="correo" class="form-label">Correo:</label>
```

```
                <input type="email" class="form-control" name="email" id="correo">
            </div>
            <div class="mb-3 col-12">
                <label for="password" class="form-label">Password:</label>
                <input type="password" class="form-control" name="password" id="password">
            </div>
            <div class="mb-3 col-12">
                <input type="submit" value="Ingresar" class="form-control btn btn-warning" />
            </div>
        </div>
    </form>
</div>
{% endblock %}
```

Observaciones:

1. No se requiere una plantilla para ‘logout’, ya que se redirecciona a ‘login’.
2. En la etiqueta <form> se define el método POST
3. Dentro del <form> es necesario insertar `{% csrf_token %}` para generar un token de seguridad exigido por Django.

IMPLEMENTANDO UN CARRITO DE COMPRAS

La siguiente imagen muestra el diseño del carrito, a implementar.

Figura 9. Carrito de compras

Las operaciones para realizar son las siguientes:

1. Ingresar un producto al carro, se realiza desde la pantalla de detalle del producto, ya está la URL en la plantilla 'producto.html'.
2. Desplegar el carrito incluyendo un resumen de los totales.
3. Aumentar o disminuir la cantidad de cada producto. Esta funcionalidad se implementará desde el lado cliente, utilizando la tecnología AJAX para comunicaciones asincrónicas cliente-servidor.
4. Eliminar un producto del carrito.
5. Ver el detalle de un producto.

CREAR MODELO DEL CARRITO DE COMPRAS

Para administra el carro de compras, se crea la tabla en 'models.py'. Agregar el siguiente código

```
from appUsuarios.models import Usuario


class Carro(models.Model):
    ESTADO_PROD =  (
        ('activo', 'activo'),
        ('comprado', 'comprado'),
        ('anulado', 'anulado'),
    )

    usuario = models.ForeignKey(Usuario, on_delete=models.CASCADE, null=False)
    producto = models.ForeignKey(Producto, on_delete=models.CASCADE, null=False)
    cantidad = models.IntegerField(null=False, default= 1)
    valUnit = models.DecimalField(max_digits=8, decimal_places=2)
    estado = models.CharField(max_length=20, choices=ESTADO_PROD, default='activo')
```

No olvide realizar la migración para que los cambios se reflejen en la BD.

CREAR RUTA DEL CARRITO DE COMPRAS

Activar los enlaces en el menú y crear la ruta:

En ‘nav.html’:

```
                <li class="nav-item">
                    <a class="nav-link" href="{% url 'carrito' %}">

                        Carrito
                    </a>
                </li>
```

En ‘urls.py’ las nuevas rutas para implementar las funcionalidades definidas:

```
urlpatterns = [
    path('categorias/', views.verCategorias, name='categorias'),
    path('productos/<str:idCategoria>', views.verProductosCategoria, name='productos'),
    path('producto/<str:idProd>', views.verProducto, name='un_producto'),
    path('carro/<str:idProd>', views.agregarCarro, name='agregarCarro'),
    path('carrito/', views.verCarrito, name='carrito'),
    path('eliminar/<str:id>', views.eliminarCarrito, name='eliminar'),
```

```
    path('cambiarCantidad/', views.cambiarCantidad),
]
```

CREAR VISTAS(VIEWS.PY)

Se Implementa la lógica para añadir, eliminar o cambiar cantidades de cada producto en el carrito. Ya se han creado las rutas que apuntan a 'agregarCarro', 'verCarrito', 'eliminarCarrito' y 'cambiarCantidad'.

En el archivo 'views.py':

```
def agregarCarro(request, idProd):
    idProd = int(idProd)
    regUsuario = request.user
    msj = None
    #leer reg del producto en Producto
    existe = Producto.objects.filter(id=idProd).exists()
    if existe:
        regProducto = Producto.objects.get(id=idProd)

        # si no existe en carrito:
        existe = Carro.objects.filter(producto=regProducto, estado= 'activo', usuario=
regUsuario).exists()
        if existe:
            # instanciar un objeto de la clase Carrito
            regCarro = Carro.objects.get(producto=regProducto, estado= 'activo', usuario= regUsuario)
            #incrementar cantidad
            regCarro.cantidad += 1
        else:
            regCarro = Carro(producto=regProducto, usuario= regUsuario, valUnit =
regProducto.precioUnitario)

        # guardar el registro
        regCarro.save()

    else:
        # dar mensaje
        msj = 'Producto no disponible'

        # redireccionar a 'verProducto'
    return verProducto(request, idProd, msj)
```

Observaciones:

1. El registro de usuario activo está contenido en el request
2. Se verifica que el id del producto corresponda a un registro de la BD
3. Se verifica si ya existe, en estado 'activo' y para este usuario, un registro en la tabla Carro.
4. Si ya existe, se aumenta en uno la cantidad.
5. Si no, se crea el registro
6. Se retorna a la pantalla del producto

```
def verCarrito(request):
    context = consultarCarro(request)
    return render(request, 'productos/carrito.html', context)
```

La lógica para ver el carrito se implementa en la función auxiliar consultarCarro(), que es usada también por cambiarCantidad()

```
def eliminarCarrito(request, id):
    #Consultar el reg y cambiar estado
    regCarrito = Carro.objects.get(id=id)
    regCarrito.estado = 'anulado'
    #guardar en BD
    regCarrito.save()
    #Desplegar el carrito
    return verCarrito(request)
```

Para eliminar un producto del carro, basta con cambiar su estado.

```
def cambiarCantidad(request):
    is_ajax = request.META.get('HTTP_X_REQUESTED_WITH') == 'XMLHttpRequest'
    if is_ajax:
        if request.method == 'POST':
            #Toma la data enviada por el cliente
            data = json.load(request)
            id = data.get('id')
            cantidad = int(data.get('cantidad'))
            if cantidad > 0:
                #Lee el registro y lo modifica
                regProducto = Carro.objects.get(id=id)
                regProducto.cantidad = cantidad
```

```
                regProducto.save()

            context = consultarCarro(request)
            return JsonResponse(context)

        return JsonResponse({'alarma': 'no se pudo modificar...'}, status=400)

    else:
        return verCarrito(request)
```

El proceso para cambiar la cantidad se inicia al modificar el valor en las etiquetas <input type='number'> del carrito. Esta parte utiliza la tecnología AJAX. Acá se observan las operaciones en el lado del servidor.

1. Se comprueba si el request es del tipo AJAX, si no, se muestra el carro.
2. Si es Ajax, se verifica el método POST.
3. Después se captura la data transmitida en formato JSON, el id y la nueva cantidad
4. Se lee el registro y se modifica la cantidad
5. Se responde al cliente con los datos actualizados del carro, en formato Json.

```
def consultarCarro(request):
    #get usuario
    regUsuario = request.user
    #filtrar productos de ese usuario en estado 'activo'
    listaCarrito = Carro.objects.filter(usuario= regUsuario, estado= 'activo').values('id',
'cantidad', 'valUnit', 'producto__imgPeque', 'producto__nombre','producto__unidad', 'producto__id')
    #renderizar
    listado = []
    subtotal = 0
    for prod in listaCarrito:
        reg = {
            'id': prod['id'],
            'cantidad':  prod['cantidad'],
            'valUnit':  prod['valUnit'],
            'imgPeque':  prod['producto__imgPeque'],
            'nombre':  prod['producto__nombre'],
            'unidad':   prod['producto__unidad'],
            'total':   prod['valUnit'] * prod['cantidad'],
            'prodId':   prod['producto__id'],
        }
        subtotal += prod['valUnit'] * prod['cantidad']

        listado.append(reg)
```

```
    envio = 8000
    if len(listado) == 0:
        envio = 0

    context = {
        'titulo': 'Productos en el carrito de compras',
        'carrito': listado,
        'subtotal': subtotal,
        'iva': int(subtotal) * 0.19,
        'envio': envio,
        'total': int(subtotal) * 1.19 + envio
    }

    return context
```

Esta función se encarga de crear el contexto con los datos del carro, incluidos los subtotales, iva, etc. Y es utilizada por cambiarCantidad() y verCarrito().

RENDERIZAR(TEMPLATES.HTML)

Crear la plantilla del carrito (‘carrito.html’) y renderizar los datos enviados en el contexto

```
{% extends 'base.html' %}
{% load static %}
{% block content %}
<!-- BLOQUE PRINCIPAL -->

<div class="py-5 text-center"><h2>{{titulo}}</h2></div>

<table class="table" style="width:90%">
    <thead>
        <tr>
            <th>Cantidad</th>
            <th></th>
            <th>Producto</th>
            <th>Precio Unit.</th>
            <th>Total</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        {% for prod in carrito %}
```

```
        <tr>
            <th><input type="number" name="cantidad" id="cantidad_{{prod.id}}"
onchange="cambiarCantidad({{prod.id}})"
                    value="{{prod.cantidad}}" min="1" style="width: 80px;" data-
preciou="{{prod.valUnit}}"></th>

            <td><img src="/media/iconos/{{prod.imgPeque}}" alt="X"></td>
            <td>{{ prod.nombre }}</td>
            <td>{{prod.valUnit}}/{{prod.unidad}}</td>
            <td id="total_{{prod.id}}">${{prod.total}}</td>
            <td>
                <a href="{% url 'eliminar' prod.id %}" class="btn btn-danger btn-carrito">Eliminar</a>
                <a href="{% url 'un_producto' prod.prodId %}" class="btn btn-info btn-
carrito">Detalle</a>
            </td>
        </tr>
        {% endfor %}

    </tbody>
</table>

<div class="col-6 offset-4 row justify-content-center border-bottom">

    <div class="col-6 text-end">
        <h6><strong> Subtotal</strong></h6>
        <h6>Iva</h6>
        <h6>Envio</h6>
        <h6><strong>A Pagar</strong></h6>
    </div>
    <div class="col-6 text-end">
        <h6><strong id="subtotal">${{ subtotal }}</strong></h6>
        <h6 id="iva">${{ iva }}</h6>
        <h6 id="envio">${{ envio }}</h6>
        <h6><strong id="total">${{ total }}</strong></h6>
    </div>

    <a href="#" class="btn btn-info btn-carrito">Pagar</a>
</div>

<!-- FIN BLOQUE PRINCIPAL -->
{% endblock %}
```

USANDO AJAX CON DJANGO

Este segmento del taller está enfocado en el conocimiento y empleo de AJAX, que permite al usuario de la aplicación web interactuar con el servidor sin tener que volver a cargar la página web. Hace uso de comunicación asincrónica entre el cliente y el servidor, lanzándose un hilo de ejecución separado que espera la respuesta del servidor sin que la página del usuario se congele.

En la carpeta 'static/js/' crear un archivo 'controlador.js' con el código Javascript del lado cliente:

```
**
 * Cambia la cantidad de un producto en el carrito
 * @param {int} id: PK del registro del produto en el carrito
 */
 function cambiarCantidad(id) {
    let cantidad = document.getElementById('cantidad_'+id).value;
    let valorUnit = document.getElementById('cantidad_'+id).dataset.preciou;

    let url = "http://localhost:8000/productos/cambiarCantidad/";
    let datos = {
        'id': id,
        'cantidad': cantidad
    };

    let total = parseFloat(cantidad) * parseFloat(valorUnit);
    document.getElementById('total_'+id).innerText = '$' + total;
    mensajeAjax(url, datos, cambiarCantidadResp)
}

function cambiarCantidadResp(data) {
    document.getElementById('subtotal').innerText = '$' + data['subtotal'];
    document.getElementById('iva').innerText = '$' + data['iva'];
    document.getElementById('envio').innerText = '$' + data['envio'];
    document.getElementById('total').innerText = '$' + data['total'];

}
```

Observaciones:

1. La función cambiarCantidad(id) es el manejador del evento 'onChange', asignado en la etiqueta de la plantilla del carro, con la PK como argumento:

```
<input type="number" name="cantidad" id="cantidad_{{prod.id}}" onchange="cambiarCantidad({{prod.id}})"
```

2. La plantilla ‘base.html’ debe tener el enlace a este archivo:

```
<script src="{% static 'js/controlador.js' %}"></script>
```

3. Desde el JS se toma la nueva cantidad desde el elemento <input>
4. El id del elemento se forma con el texto ‘cantidad_’ mas la PK, esto se hace en la plantilla
5. Para cambiar el nuevo subtotal en esta fila del carrito, es necesario conocer el precio unitario, que se pasa como un atributo de usuario desde la plantilla

```
        data-preciou="{{prod.valUnit}}"
```

6. y se lee en JS con

```
let valorUnit = document.getElementById('cantidad_'+id).dataset.preciou;
```

7. La URL destino es igual al compuesto por las URL del lado Servidor, no se puede usar los ‘name’ de las rutas.
8. Se ensamblan los datos a enviar al servidor por medio de la función mensajeAjax()
9. Y se modifica el subtotal.
10. La función cambiarCantidadResp() es la que se ejecuta de manera asincrónica y se encarga de actualizar los totales, que son enviados desde el servidor.

El código para implementar la comunicación AJAX es el siguiente:

```
/**
 * Consulta AJAX al servidor por método POST
 * @param {*} urlserver :Direccion de envio
 * @param {*} datos     :Data en formato JavaScript object
 * @param {*} callBackFunction : Funcion de retorno
 */
function mensajeAjax(urlserver, datos, callBackFunction) {

   const csrftoken = getCookie('csrftoken');
   fetch(urlserver, {
       method: 'POST',
       credentials: 'same-origin',
       headers: {
```

```
            'Accept': 'application/json',
            'X-Requested-With': 'XMLHttpRequest',
            'X-CSRFToken': csrftoken,
        },
        body: JSON.stringify(datos) //JavaScript object of data to POST
    })
        .then(response => response.json())  //Convierte la respuesta JSON en data
        .then(data => {
            //mostrarAviso(data)
            callBackFunction(data)
        })
        .catch((error) => {
            console.error('Error:', JSON.stringify(error));
        });
}
```

Por seguridad implementada por Django, es necesario recuperar la cookie con el token de seguridad en el navegador:

```
/**
 * @param {*} name Nombre de la cookie
 * @returns el cvontenido de la cookie
 */
function getCookie(name) {
    let cookieValue = null;
    if (document.cookie && document.cookie !== "") {
        const cookies = document.cookie.split(";");
        for (let i = 0; i < cookies.length; i++) {
            const cookie = cookies[i].trim();
            // Does this cookie string begin with the name we want?
            if (cookie.substring(0, name.length + 1) === (name + "=")) {
                cookieValue = decodeURIComponent(cookie.substring(name.length + 1));
                break;
            }
        }
    }
    return cookieValue;
}
```

ENVÍO DE CORREOS-E

CONFIGURACIÓN DEL CORREO

Para practicar el envío de correos, vamos a activar el botón 'Pagar' del carrito; pero antes de eso usted debe tener un correo configurado para permitir envíos desde un servidor, por favor seguir los pasos del Anexo 4 para configurar un correo Gmail[3].

CREAR LA RUTA

Después de configurado el correo y agregadas las variables en 'settings.py', como se explica en el anexo 4, se procede a construir la ruta.

1. ACTIVAR URL EN EL ENLACE(HTML)

En la plantilla 'carrito.html' agregar

```
    <a href="{% url 'pagar' %}" class="btn btn-info btn-carrito">Pagar</a>
```

2. CREAR PATH(URLS.PY)

En 'urls.py' de appProductos agregar la ruta

```
urlpatterns = [
   … otras rutas …
    path('pagar/', views.pagarCarrito, name='pagar'),
]
```

3. CREAR VISTA(VIEWS.PY)

Agregar la función 'pagarCarrito()'

```
…
from django.template.loader import render_to_string
from django.core.mail import EmailMessage
…
```

[3] Atención: No utilice el corre institucional '@misena.edu.co', pues será bloqueado.

```python
def pagarCarrito(request):
    context = consultarCarro(request)
    regUsuario = request.user
    nombreUsuario = str(regUsuario)
    context['nombre'] = nombreUsuario
    correo = regUsuario.email

    #--- MODULO PARA ENVIO DE CORREO
    mail_subject = 'Factura de compra'

    body = render_to_string('productos/html_email.html', context)
    to_email = [correo]   #Lista con el o los correos de destino

    send_email = EmailMessage(mail_subject, body, to= to_email )
    send_email.content_subtype = 'html'
    send_email.send()
    #---FIN MODULO PARA ENVIO DE CORREO DE CONFIRMACION

    # sacar productos del carrito
    listaCarrito = Carro.objects.filter(usuario= regUsuario, estado= 'activo')
    for regCarro in listaCarrito:
        regCarro.estado = 'comprado'
        regCarro.save()

    #redireccionar
    return verCategorias(request)
```

Observaciones al código anterior:

1. Se utiliza la función consultarcarro() para crear el context con toda la información del carrito de compras.
2. Se la agrega al diccionario context el nombre del usuario.
3. Se definen los argumentos necesarios para el correo: asunto, contenido, destinatario
4. Se crea un objeto de la clase ‘EmailMessage’ y se cambia su atributo ‘content_subtype’ para aceptar etiquetas HTML.
5. Después de enviado el correo, se procede a cambiar el estado de los productos comprados, para que no sigan apareciendo en en el carrito.
6. Se redirecciona a otra página para continuar el proceso de compra.

4. RENDERIZANDO EL HTML EN EL CORREO

Crear la plantilla ‘html_email.html’:

```
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">

<head>
    <style>
        table {
            font-family: arial, sans-serif;
            border-collapse: collapse;
            width: 80%;
            border: none;
            font-size: 14px;
            line-height: 1.5;
        }

        td,
        th {
            border: 1px solid #dddddd;
            text-align: left;
            padding: 8px;
        }

        tr:nth-child(even) {
            background-color: #cebebe;
        }

        #totales {
            width: 30%;
        }
    </style>
</head>

<body>
    <h4>
        Estimado {{nombre}}:<br>Gracias por su compra, la siguiente es la relación de productos a
despachar:
    </h4>
    <br>
    <table align="center" border="0" cellpadding="0" cellspacing="0" width="250" style="">
        <tr>
            <th>Cantidad</th>
```

```
            <th>Producto</th>
            <th>Precio Unit.</th>
            <th>Total</th>

        </tr>

        {% for prod in carrito %}
        <tr>
            <td>{{prod.cantidad}}</td>
            <td>{{ prod.nombre }}</td>
            <td>{{prod.valUnit}}/{{prod.unidad}}</td>
            <td>${{prod.total}}</td>
        </tr>
        {% endfor %}
    </table>
    <br>
    <div style="">
        <table align="center" border="0" cellpadding="0" cellspacing="0" id="totales">
            <tr>
                <th>Subtotal</th>
                <td>${{ subtotal }}</td>
            </tr>
            <tr>
                <th>IVA</th>
                <td>${{ iva }}</td>
            </tr>
            <tr>
                <th>Envio</th>
                <td>${{ envio }}</td>
            </tr>
            <tr>
                <th>A pagar</th>
                <td>${{ total }}</td>
            </tr>
        </table>
    </div>
</body>
</html>
```

Observe que el odigo HTML a ser enviado en un correo, debe tener los estilos dentro de la etiqueta <style>.

La anterior plantilla se renderiza en el cuerpo de un correo como se muestra en la siguiente figura.

mleonpruebas@gmail.com
para mí

21:49 (hace 23 minutos)

Estimado Marco Mora:
Gracias por su compra, la siguiente es la relación de productos a despachar:

Cantidad	Producto	Precio Unit.	Total
1	Ajo	1200.00/Libra	$1200.00
1	Agua	2000.00/Frasco x 100 ml	$2000.00

Subtotal	$3200.00
IVA	$608.0
Envio	$8000
A pagar	$11808.0

Figura 10. Correo con la descripción de la compra

El último bloque del taller, referido a Django, hace uso de los 'forms' o formularios para entrada de datos, predefinidos a partir del modelo.

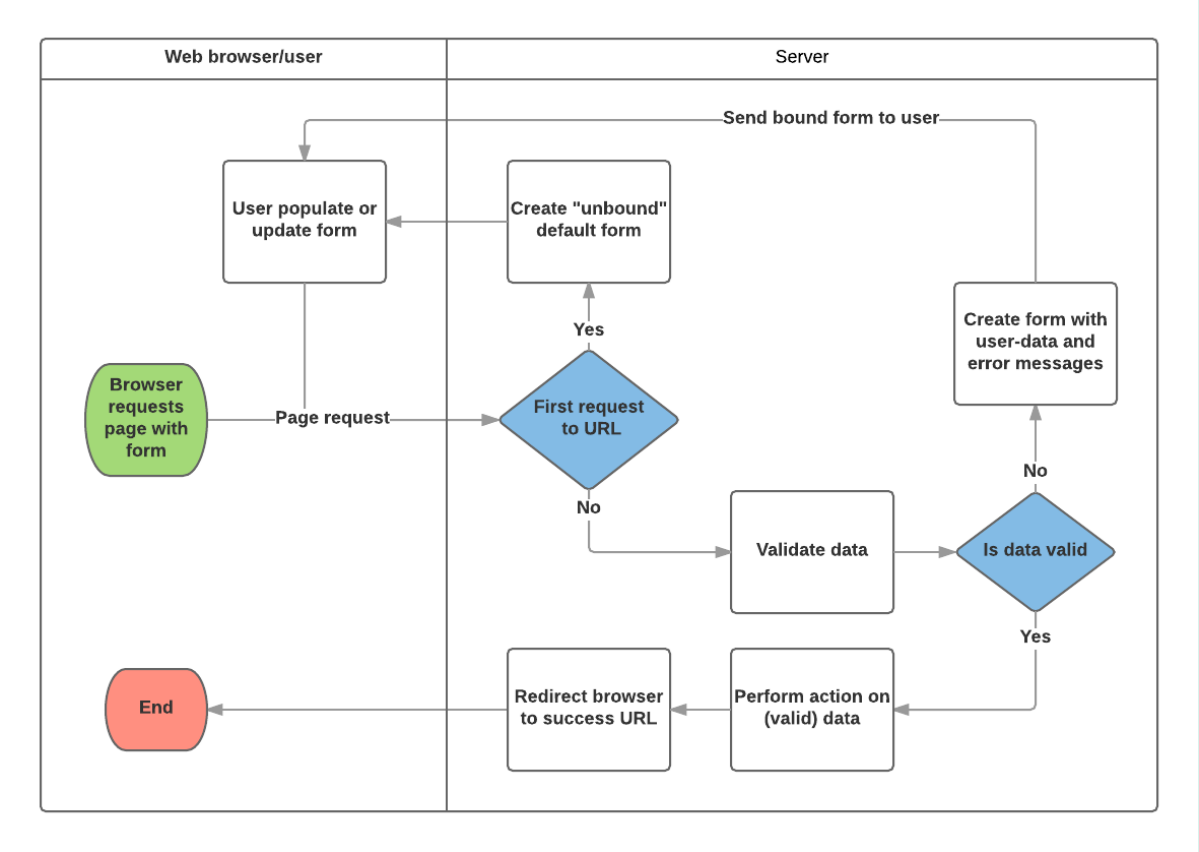


Figura 11. Diagrama de flujo de un Form

En la figura anterior[4] se presenta el diagrama de flujo de un 'form' creado a partir la clase Form, opcionalmente asociado a una clase de Model (una tabla):

1. El usuario hace el requerimiento (Page request)
2. Si es la primera vez, normalmente por el método GET, se crea un objeto de la clase form.
3. El usuario diligencia el formulario y lo envía de regreso al servidor (Submit)
4. Si no es la primera vez (método POST) el servidor valida los datos
5. Si los datos NO son correctos, se vuelve a crear el objeto form, incluyendo los datos enviados por el usuario y los mensajes de error y se envían al cliente.

[4] Fuente de la figura: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Forms

6. Si los datos son correctos, se ejecuta la lógica necesaria con los datos recibidos y se redirecciona a otra página

REGISTRO DE USUARIOS

Para esta práctica, vamos a realizar un registro de usuarios. Se quiere que cada usuario cliente pueda ingresar sus datos para el registro en la aplicación.

1. ACTIVAR URL EN EL ENLACE(HTML)

En la plantilla ‘nav.html’ agregar:

```
                <li class="nav-item">
                    <a class="nav-link" href="{% url 'registro' %}">
                        Registrarse
                    </a>
                </li>
```

2. CREAR PATH(URLS.PY)

Agregar la ruta en ‘urls.py’ de ‘appUsuarios’

```
   path('registro/', views.registrar, name='registro'),
```

3. CREAR EL FORM (FORMS.PY)

En la appUsuarios cree el archivo ‘forms.py’

```
from django import forms
from .models import Usuario

class UsuarioForm(forms.ModelForm):
    class Meta:
        model = Usuario
        fields = ['first_name', 'last_name', 'username', 'email']
```

Observaciones:

1. La clase ‘UsuarioForm’ hereda de forms.ModelForm

2. La clase Meta dentro de la definición de clase del form, se utiliza para adjuntar metadatos (en este caso la clase Usuario desde Models)
3. Se puede definir una lista de campos a mostrar en el formulario

4. CREAR VISTA(VIEWS.PY)

Inicialmente se ingresa el código para desplegar el formulario:

```
…
from .forms import UsuarioForm
…

def registrar(request):
    if request.method == 'POST':
        pass
    else:
        form = usuarioForm()
        return render(request, 'usuarios/registro.html', {'form': form})
```

Observe que el objeto ‘form’ es una instancia de la clase UsuarioForm creada en el paso anterior

5. RENDERIZANDO EL HTML

Cree la plantilla ‘registro.html’ dentro de ‘templates/usuarios/’

```
{% extends 'base.html' %}
{% block content %}
<br>
<div class="mb-3 descripcion">
    <h4>Registro de Usuario</h4>
</div>
<div class="mb-3">
    <form enctype="multipart/form-data" action="{% url 'registro' %}" method="POST">
        {% csrf_token %}
         {{ form }}
        <br>
        <div class="row justify-content-end">
            <div class="mb-3 col-6">
                <input type="submit" value="Registarse en el Sistema" class="form-control btn btn-
warning" />
            </div>
        </div>
```

```
    </form>
</div>
{% endblock %}
```

El formulario se presenta como en la figura siguiente:

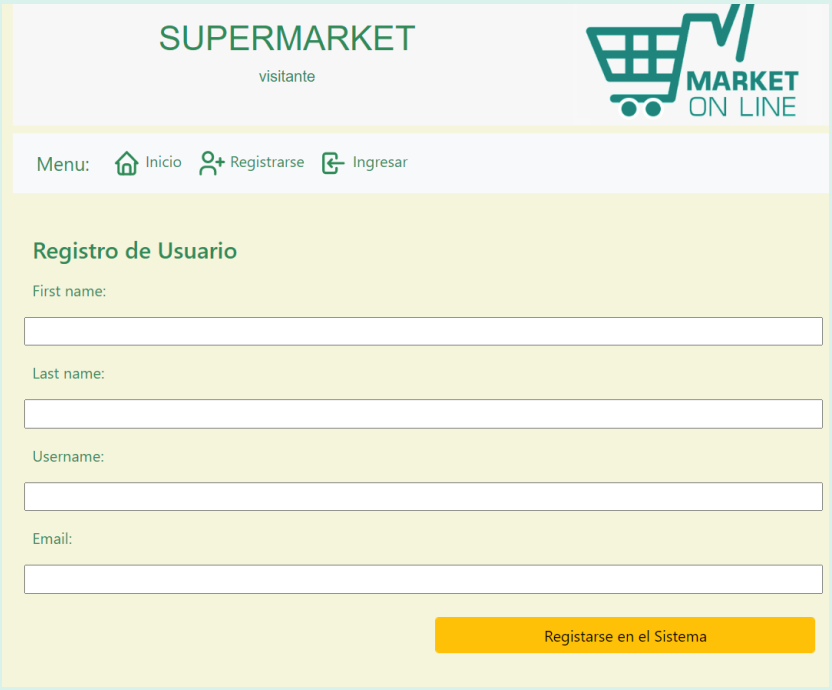


Figura 12. Formulario de registro sin aplicación de clases CSS

RENDERIZAR FORMS USANDO WIDGET-TWEAKS

El formulario anterior no tiene una apariencia similar a los formularios con clases Bootstrap, Para mejorar la presentación se hace uso de la librería Python-widget-tweaks que permite agregarle estilos al formulario creado.

1. Desmonte el servidor e instale la librería desde la consola

```
(env)
mlmor@MarcoLeon MINGW64 ~/OneDrive/Documents/SENA/ADSI
p (main)
$ pip install django-widget-tweaks
```

2. Registre como una nueva APP en ‘settings.py’:

```
INSTALLED_APPS = [
… etc
    'appProductos',
    'appUsuarios',
    'widget_tweaks',
]
```

3. Cambie la plantilla ‘registro.html’, así:

```
{% extends 'base.html' %}
{% block content %}
<br>
<div class="mb-3 descripcion">
    <h4>Registro de Usuario</h4>
</div>
<div class="mb-3">
    <form enctype="multipart/form-data" action="{% url 'registro' %}" method="POST">
        {% csrf_token %}
        {% load widget_tweaks %}
        <div class="row g-3">
            {% for field in form %}
            <div class="col-12 form-group">
                <label for="{{ field.id_for_label }}" class="form-label">{{ field.label }}</label>
                {% render_field field class="form-control" %}
            </div>
            {% endfor %}
        </div>
        <br>
        <div class="row justify-content-end">
            <div class="mb-3 col-6">
                <input type="submit" value="Registarse en el Sistema" class="form-control btn btn-
warning" />
            </div>
        </div>
    </form>
</div>
{% endblock %}
```

En la siguiente figura, se observa el resultado al agregar clases Bootstrap, haciendo usa de la librería widget-tweaks.

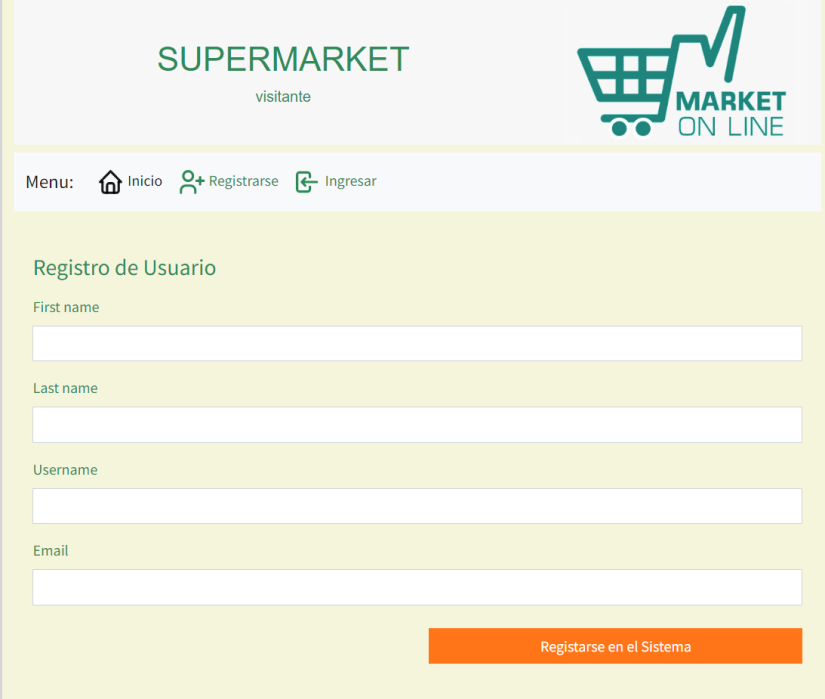


Figura 13. Formulario de registro con clases CSS, usando widget-tweaks

CAMBIANDO LAS ETIQUETAS DEL FORMULARIO

Si se desea cambiar las etiquetas por defecto (Django toma el nombre de las columnas en la tabla, iniciando con Mayúscula y reemplaza la línea al piso por espacio), agregar a 'forms.py':

```
from django import forms
from .models import Usuario

class UsuarioForm(forms.ModelForm):

    class Meta:
        model = Usuario
        fields = ['first_name', 'last_name', 'email', 'password']
        labels = {
            'first_name': 'Nombre: ',
            'last_name': 'Apellido: ',
            'email': 'Correo E: ',
            'password': 'Contraseña: ',
        }
```

Con el siguiente resultado:

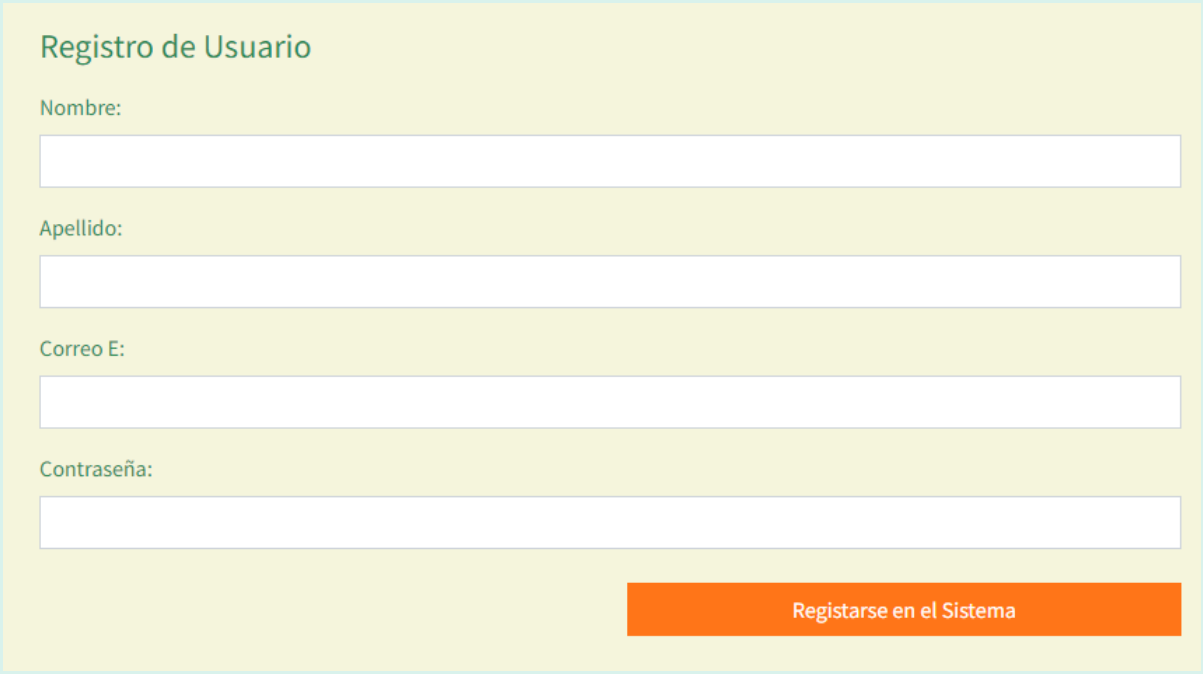


Figura 14. Agregando etiquetas personalizadas

COMPLETANDO EL REGISTRO DE USUARIOS (VIEWS.PY)

Se implementa el código para crear nuevo usuario a partir de los datos enviados por el visitante, completando la función registrar() cuando el método es POST:

```
def registrar(request):
    if request.method == 'POST':
        form = UsuarioForm(request.POST)
        if form.is_valid():
            first_name = form.cleaned_data['first_name']
            last_name = form.cleaned_data['last_name']
            email = form.cleaned_data['email']
            password = form.cleaned_data['password']

            username = email.split('@')[0]

            existe = Usuario.objects.filter(email=email).exists()
```

```
            if not existe:
                user = Usuario.objects.create_user(first_name=first_name, last_name=last_name,
username=username, email=email, password=password)
                user.is_active = True
                user.save()
                return render(request, 'usuarios/login.html')
        else:
            return render(request, 'usuarios/registro.html', {'form': form})

    else:
        form = UsuarioForm()
        return render(request, 'usuarios/registro.html', {'form': form})
```

Observaciones:

1. Se crea el objeto ‘form’ de la clase UsuarioForm, con los datos del formulario recibido, como argumentos.
2. Si los los datos del formulario son válidos, se extraen en sendas variables.
3. El username se crea con la primera parte del email
4. Se verifica que no exista un usuario con el mismo correo
5. Si no existe, se crea el usuario y se activa inmediatamente (no recomendado en producción)
6. Si los datos no son válidos, se regresa a la página con el formulario recibido

ANEXOS

ANEXO 1. EJEMPLO DE PLANTILLA BASE

La plantilla base contiene las etiquetas HTML comunes a todas las páginas, especialmente los enlaces a archivos estático y se utiliza como plantilla inicial para 'inyectar' los bloques que se requieran.

```
{% load static %}
<!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>WorkShop Django</title>
    <link rel="stylesheet" href="{% static 'css/bootstrap.min.css' %}">
    <link rel="stylesheet" href="{% static 'css/estilo.css' %}">
    <script src="{% static 'js/bootstrap.min.js' %}"></script>
    <script src="{% static 'js/controlador.js' %}"></script>
</head>
<body>
    <div class="container">
        <div class="row">
            <div class="contenedor col-12 col-md-8 offset-md-2">
                {% include 'includes/head.html' %}
                {% include 'includes/nav.html' %}

                <div class="contenido">
                    {% block content %}

                    {% endblock %}
                </div>
                {% include 'includes/footer.html' %}
            </div>
        </div>
    </div>
</body>
</html>
```

ANEXO 2. SCRIPT DE CREACIÓN DE PRODUCTOS

Para facilitar las pruebas y la visualización de las diferentes plantillas, se puede hacer uso de los siguientes scripts para, directamente en la BD, usando la herramienta SQLiteStudio u otra,poblar las tablas con datos de ejemplo.

TABLA CATEGORÍAS

```
INSERT INTO appProductos_categoria VALUES
('Alimentos'),
('Aliños'),
('Aseo'),
('Bebidas'),
('Comidas Rápidas'),
('Frutas'),
('Licor'),
('Medicamentos')
```

TABLA PRODUCTOS

```
INSERT  INTO appProductos_producto (nombre, descripcion, categoria_id, precioUnitario, unidad, existencia, imgGrande, imgPeque)
VALUES
('Banana', 'Fruta tropical', 6, 2500, 'Libra', 32, 'productos/1banano.jpg', 'iconos/1banano.jpg'),
('Curuba', 'Citrico', 6, 3000, 'Libra', 13, 'productos/1curuba.jpg', 'iconos/1curuba.jpg'),
('Fresa', 'Fruto rojo', 6, 2800, 'Libra', 7, 'productos/1fresa.jpg', 'iconos/1fresa.jpg'),
('Vodka', 'Bebida Rusa', 6, 35000, 'Und x 375cc', 56, 'productos/absolut375.jpg', 'iconos/absolut375.jpg'),
('Cerveza', 'Bebida Boyaca', 6, 25000, 'sixpack lata', 8, 'productos/aguilacerobotsix.jpg', 'iconos/aguilacerobotsix.jpg'),
('Cerveza', 'Bebida Boyaca', 6, 12000, 'lata', 87, 'productos/aguilalight330.jpg', 'iconos/aguilalight330.jpg'),
('Cerveza', 'Bebida Boyaca', 6, 450000, 'Tetrapack x 300ml', 45, 'productos/aguardiente.jpg', 'iconos/aguardiente.jpg'),
('Acetaminofen', 'El paracetamol o acetaminofén', 8, 5000, 'Caja x 100', 24, 'productos/acetaminofen.jpg', 'iconos/acetaminofen.jpg'),
('Amoxicilina', 'Antibiotico', 8, 1000, 'Und', 98, 'productos/amoxicilina.jpg', 'iconos/amoxicilina.jpg'),
('Advilmax', 'Ibufrofeno', 8, 500, 'Sobre x 3', 5, 'productos/advilmax.jpg', 'iconos/advilmax.jpg'),
('AdvilUltra', 'Ibufrofeno', 8, 780, 'Caja x 50', 23, 'productos/advilultra.jpg', 'iconos/advilultra.jpg'),
('Acohol', 'Topico', 8, 12000, 'Frasco x 150 ml', 54, 'productos/alcohol.jpg', 'iconos/alcohol.jpg'),
```

('Algodon', 'Algodón', 8, 4000, 'Paquete x 50gm', 98, 'productos/algodon.jpg', 'iconos/algodon.jpg'),

('Agua', 'Agua', 4, 2000, 'Frasco x 100 ml', 56, 'productos/aguaox.jpg', 'iconos/aguaox.jpg'),

('Ahuyama', 'Hortaliza', 1, 1500, 'Libra', 10, 'productos/ahuyama.jpg', 'iconos/ahuyama.jpg'),

('Ajo', 'Hortaliza', 1, 1200, 'Libra', 44, 'productos/ajo.jpg', 'iconos/ajo.jpg'),

('AlitasPollo', 'Carnes', 1, 6000, 'Libra', 85, 'productos/alaspollo.jpg', 'iconos/alaspollo.jpg'),

('Albondigasx6', 'Carnes', 1, 6500, 'Und x 162gm', 46, 'productos/albondigas162.jpg', 'iconos/albondigas162.jpg'),

('AlbondigasX12', 'Carnes', 1, 11900, 'Und x 322gm', 93, 'productos/albondigas322.jpg', 'iconos/albondigas322.jpg'),

('Alpicrema', 'Bebida lactosada', 1, 4500, 'Tetrapack x 300ml', 24, 'productos/alpinacrema200.jpg', 'iconos/alpinacrema200.jpg'),

('Alpinades', 'Bebida Azucarada', 1, 3000, 'Bolsa x350cc', 17, 'productos/alpinades1100.jpg', 'iconos/alpinades1100.jpg'),

('AlpinaHolandesa', 'Queso', 1, 8700, 'Unidad x 450gm', 48, 'productos/alpinaholandes450.jpg', 'iconos/alpinaholandes450.jpg'),

('AlpinaMelo', 'Yogourt Desalactosado', 1, 2560, 'Bolsa x350cc', 94, 'productos/alpinameloc900.jpg', 'iconos/alpinameloc900.jpg'),

('AlpinaVaso', 'Avena', 1, 1780, 'Vaso x150cc', 23, 'productos/alpinavaso.jpg', 'iconos/alpinavaso.jpg'),

('AlqueriaB1100', 'Leche', 1, 500, 'Bolsa x350cc', 43, 'productos/alqueria1100.jpg', 'iconos/alqueria1100.jpg'),

('AlqueriaD900', 'Leche', 1, 780, 'Bolsa x350cc', 55, 'productos/alqueriades900.jpg', 'iconos/alqueriades900.jpg'),

('ArepaChorizo', 'carbohidratos y carne', 5, 12000, 'Und', 7, 'productos/arepachorizo.jpg', 'iconos/arepachorizo.jpg'),

('ArepaMixta', 'carbohidratos y carne', 5, 4000, 'Und', 23, 'productos/arepamixta.jpg', 'iconos/arepamixta.jpg'),

('ArepaQueso', 'carbohidratos y lacteo', 5, 2000, 'Und', 36, 'productos/arepaqueso.jpg', 'iconos/arepaqueso.jpg'),

('ArrozCamarones', 'Cereal y pescado', 5, 1500, 'Und', 36, 'productos/arrozcamarones.jpg', 'iconos/arrozcamarones.jpg'),

('ArrozPollo', 'Cereal y pollo', 5, 1200, 'Und', 76, 'productos/arrozconpollo.jpg', 'iconos/arrozconpollo.jpg'),

('Arveja', 'Cereal en Lata', 1, 6000, 'Und', 36, 'productos/arveja300.jpg', 'iconos/arveja300.jpg'),

('Bofe', 'Visceras', 1, 6500, 'Und', 21, 'productos/bofe.jpg', 'iconos/bofe.jpg'),

('BurgerJovenes', 'Procesados', 5, 11900, 'Und', 76, 'productos/burgerguey.jpg', 'iconos/burgerguey.jpg'),

('BurgerNiños', 'Procesados', 5, 4500, 'Und', 34, 'productos/burgerkids.jpg', 'iconos/burgerkids.jpg'),

('Burrito', 'Procesados', 5, 3000, 'Und', 45, 'productos/burrito1.jpg', 'iconos/burrito1.jpg'),

('Callo', 'Visceras', 1, 8700, 'Und', 23, 'productos/callo.jpg', 'iconos/callo.jpg'),

('CanelaMolida', 'Especia', 1, 2560, 'Und', 76, 'productos/canelamolida30.jpg', 'iconos/canelamolida30.jpg'),

('AcieteCanola', 'semillas oleaginosas', 1, 4000, 'Und', 98, 'productos/canola2000.jpg', 'iconos/canola2000.jpg'),

('Champiñon', 'hongo', 1, 2000, 'Und', 100, 'productos/champinon.jpg', 'iconos/champinon.jpg'),

('Chatas', 'Carnes', 1, 1500, 'Und', 23, 'productos/chatares.jpg', 'iconos/chatares.jpg'),

('Chorizo', 'Embutidos', 5, 1200, 'Und', 84, 'productos/chorizococtel.jpg', 'iconos/chorizococtel.jpg'),

('Churrasco', 'Carnes', 1, 6000, 'Unidad x 450gm', 28, 'productos/churrasco.jpg', 'iconos/churrasco.jpg'),

('CocaCola', 'Bebida Azucarada', 4, 6500, 'Und', 36, 'productos/cocacola.jpg', 'iconos/cocacola.jpg'),

('ClubDorada', 'Bebida Boyacos', 6, 11900, 'Und', 76, 'productos/clubdorada330.jpg', 'iconos/clubdorada330.jpg'),

('Hamburguesa', 'Procesados', 5, 4500, 'Und', 36, 'productos/classicburguer.jpg', 'iconos/classicburguer.jpg'),

('ClubRoja', 'Bebida Boyacos', 6, 3000, 'Und', 21, 'productos/clubrojasix.jpg', 'iconos/clubrojasix.jpg'),

('CoditoCerdo', 'Carnes', 1, 8700, 'Und', 76, 'productos/coditocerdo.jpg', 'iconos/coditocerdo.jpg'),

('DeditoCerdo', 'Carnes', 1, 2560, 'Und', 34, 'productos/detoditoxl250.jpg', 'iconos/detoditoxl250.jpg'),

django

('Distran', 'Descongestionante', 8, 1200, 'Und', 45, 'productos/dristan.jpg', 'iconos/dristan.jpg'),

('SalsaTomate', 'Ultraprocesado', 1, 6000, 'Und', 23, 'productos/frucotomate600.jpg', 'iconos/frucotomate600.jpg'),

('Champu', 'Anticaspa', 1, 6500, 'Und', 76, 'productos/head700.jpg', 'iconos/head700.jpg'),

('Lulo', 'Citrico', 6, 11900, 'Und', 98, 'productos/lulofruta.jpg', 'iconos/lulofruta.jpg'),

('Melon', 'Fruta de verano', 6, 4500, 'Und', 100, 'productos/melon.jpg', 'iconos/melon.jpg'),

('MigasCerdo', 'Procesados', 5, 3000, 'Und', 23, 'productos/migascerdo.jpg', 'iconos/migascerdo.jpg'),

('Yodora', 'Topico', 1, 8700, 'Und', 84, 'productos/yodora25gr.jpg', 'iconos/yodora25gr.jpg'),

('Zucaritas', 'Mark Zuckerberg', 1, 2560, 'Und', 28, 'productos/zucaritas730.jpg', 'iconos/zucaritas730.jpg')

ANEXO 3. GESTIONANDO LOS ARCHIVOS DE IMÁGENES

Para indicar a Django que gestione el manejo de imágenes, incluyendo su cargue y visualización, seguir los siguientes pasos

1. Crear una carpeta 'media' a nivel de la carpeta inicial, para el cargue de las imágenes
2. Agregar las variables a settings.py para registrar la carpeta y la ruta.

```
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
```

3. En la consola, bajar el servidor local: <CTRL> + C
4. Agregar la librería pillow para el manejo de imágenes:

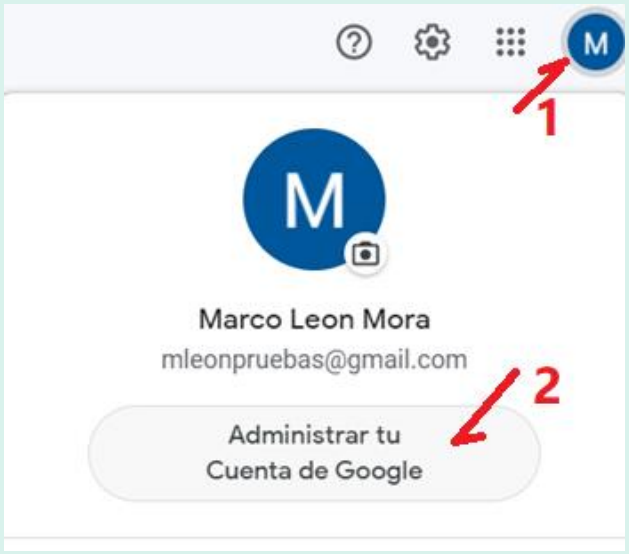
```
mlmor@MarcoLeon MINGW64 ~/OneDrive/Documents/SENA/ADSI 2022/TallerDjango/
p (main)
$ pip install pillow
```

5. En 'urls.py' del proyecto principal, agregar:

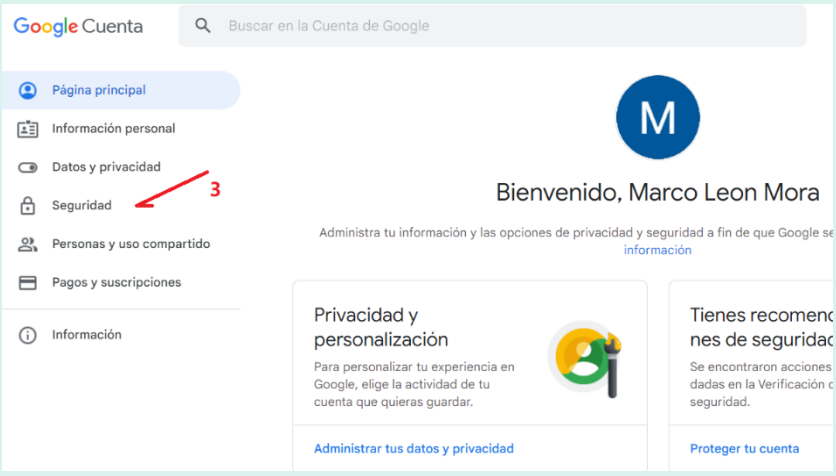
```
from django.contrib import admin
from django.urls import path, include
from . import views

from django.conf.urls.static import static
from django.conf import settings

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.home, name='home'),
    path('productos/', include('appProductos.urls')),
]+ static(settings.MEDIA_URL, document_root = settings.MEDIA_ROOT)
```

ANEXO 4. ACTIVACION DE CORREO GMAIL PARA ENVIO DESDE UNA APP WEB CON DJANGO

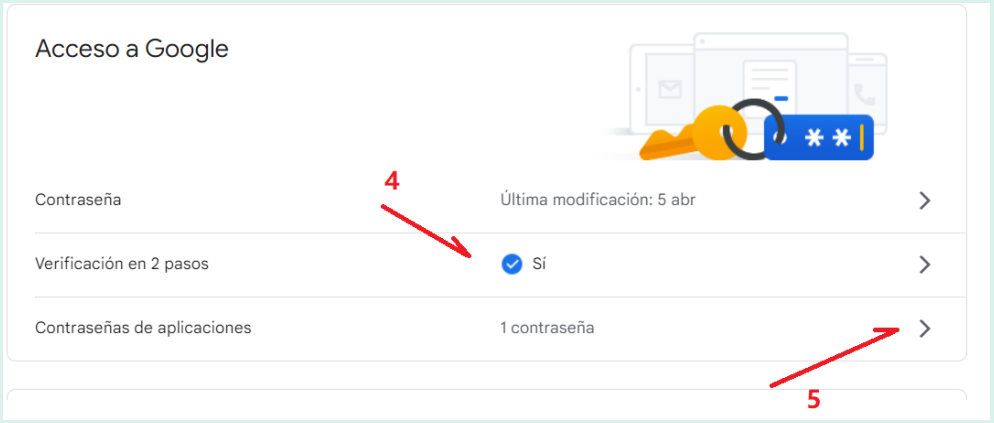
1. Abrir el correo gmail, clic en la imagen del usuario (esquina superior derecha)
2. Luego en 'Administrar tu Cuenta de Google'



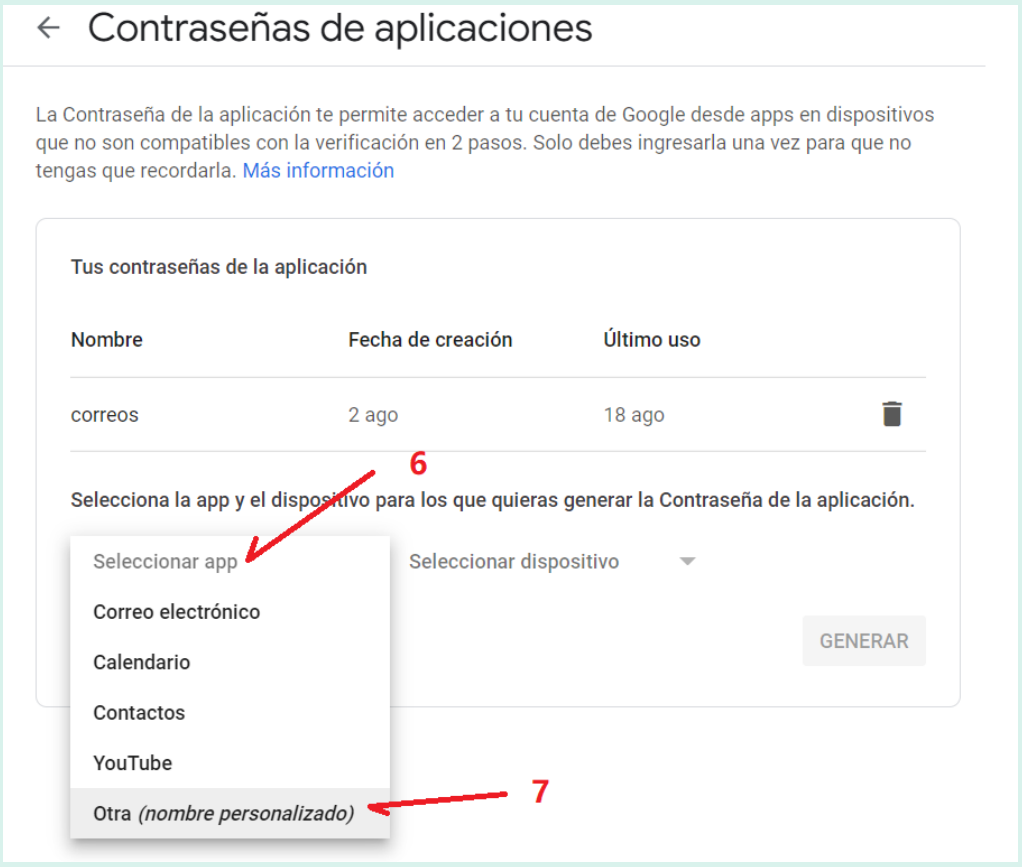
3. Clic en 'Seguridad' (menú a la izquierda)



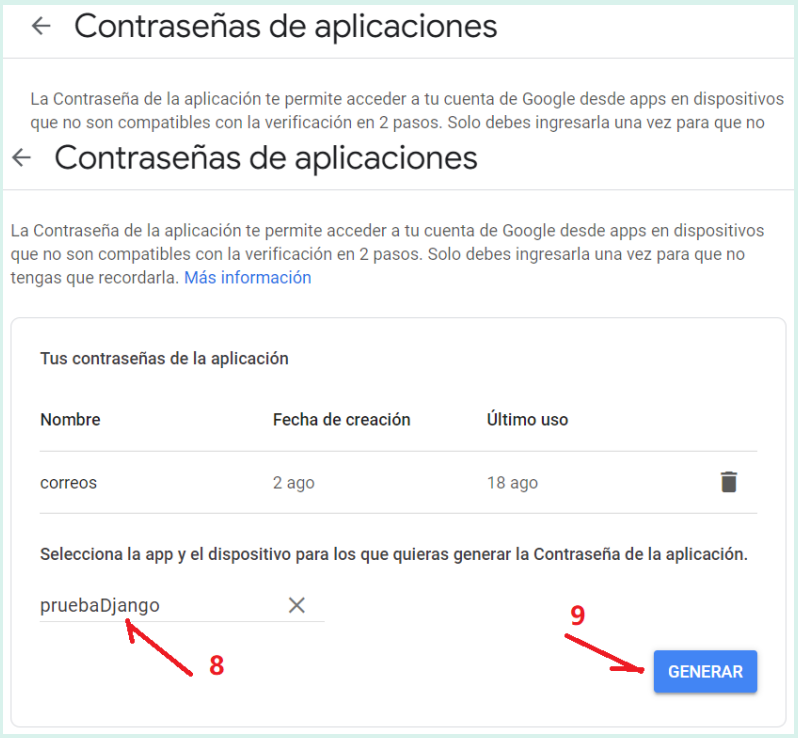
4. En la sección 'Acceso a Google', activar 'Verificación en dos pasos'
5. Abrir 'Contraseña de aplicaciones'



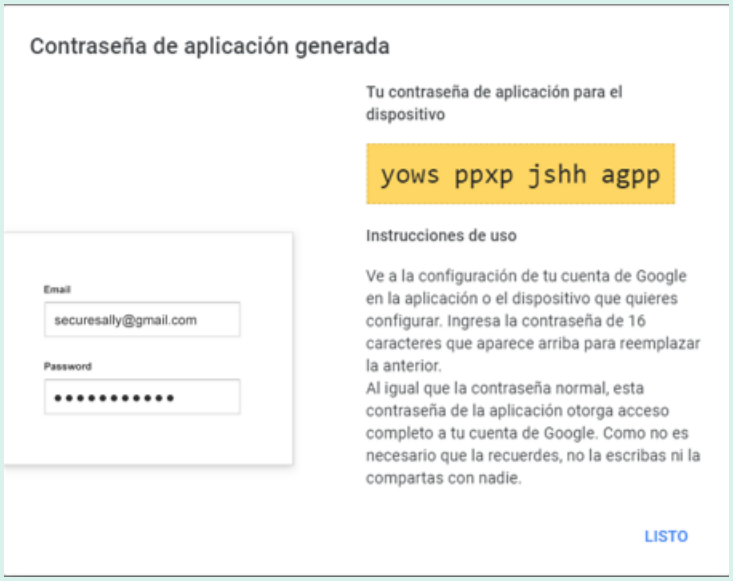
6. Clic en ‘Seleccionar app’
7. Activar ‘Otra…’



8. Asignar un nombre a la nueva Contraseña de aplicaciones
9. Clic en el botón ‘Generar’



10. Copiar la contraseña generada en el recuadro amarillo



11. En el archivo de configuración del proyecto Django, insertar las siguientes líneas

```
#*************** EMAIL ************************************
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_PORT = 587
EMAIL_HOST_USER = 'mleonpruebas@gmail.com'
EMAIL_HOST_PASSWORD = '[redacted]'
EMAIL_USE_TLS = True
```

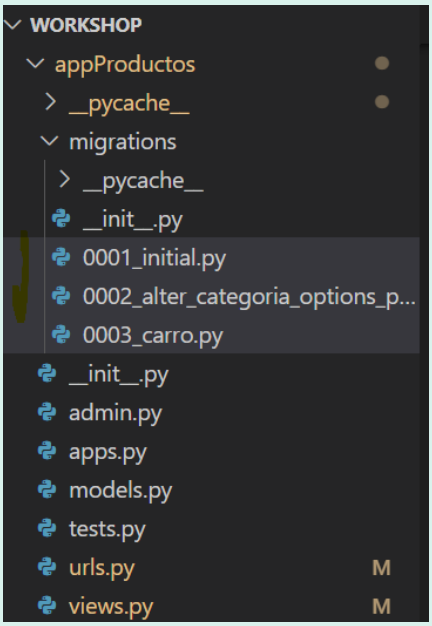
En la variable EMAIL_HOST_USER, su correo completo

En la variable EMAIL_HOST_PASSWORD, la contraseña copiada en el paso 10.

ANEXO 5. RESET DATA

Pasos necesarios para reiniciar la BD cuando las características de los cambios efectuados en los modelos impiden la migración.

1. Bajar el servidor local (<CTRL> + C)
2. Elimine todos los archivos de migración dentro de cada app. (inician con números como 0001_, etc).
3. Elimine el archivo 'db.sqlite3'
4. Lanzar el servidor para crear un nuevo archivo 'db.sqlite3'
5. Bajar el servidor de nuevo
6. En la consola ejecute los comandos de migración
    a. python manage.py makemigrations
    b. python manage.py migrate
7. Crear de nuevo el superusuario:
    a. winpty python manage.py createsuperuser



QUE ES EL MAPEO OBJETO-RELACIONAL ORM

El Mapeo Objeto-Relacional o como se conocen comúnmente, ORM (del inglés Object Relational Mapping) permite convertir los datos de los objectos en un formato correcto para poder guardar la información en una RDB o base de datos relacional. El mapeo crea una base de datos virtual donde los datos que se encuentran en la aplicación (Objetos) quedan vinculados a la base de datos (Relacional), permitiendo la persistencia.

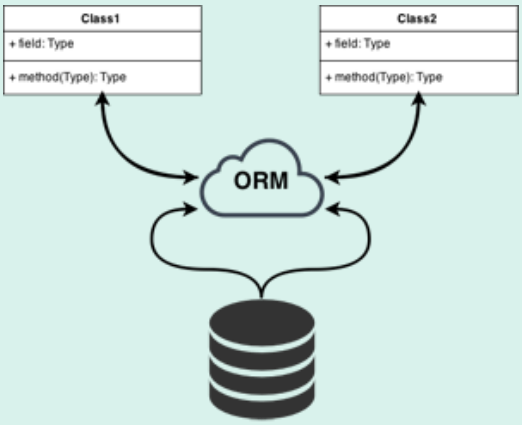


Figura 15. Mapeo Objeto-Relacional[5]

Cuando se programa alguna aplicación que se conecta a una base de datos, es laborioso transformar toda la información que se recibe desde la base datos, principalmente en tablas, a los objetos de la aplicación y viceversa. A esto se le denomina MAPEO. Utilizando un ORM este mapeo será automático, e independiente de la base de datos utilizada en ese momento, lo que permite un rápido cambio del motor BD.

METODOS QUERY

Algunas funciones del ORM de Django y filtros para implementar consultas complejas a la BD

Method	Description
filter()	Filter by the given lookup parameters. Multiple parameters are joined by SQL AND statements (See Chapter 4)
exclude()	Filter by objects that don't match the given lookup parameters

[5] Tomado de https://programarfacil.com/blog/que-es-un-orm/

annotate()	Annotate each object in the QuerySet. Annotations can be simple values, a field reference or an aggregate expression
order_by()	Change the default ordering of the QuerySet
reverse()	Reverse the default ordering of the QuerySet
distinct()	Perform an SQL SELECT DISTINCT query to eliminate duplicate rows
values()	Returns dictionaries instead of model instances
values_list()	Returns tuples instead of model instances
dates()	Returns a QuerySet containing all available dates in the specified date range
datetimes()	Returns a QuerySet containing all available dates in the specified date and time range
none()	Create an empty QuerySet
all()	Return a copy of the current QuerySet
union()	Use the SQL UNION operator to combine two or more QuerySets
intersection()	Use the SQL INTERSECT operator to return the shared elements of two or more QuerySets
difference()	Use the SQL EXCEPT operator to return elements in the first QuerySet that are not in the others
select_related()	Select all related data when executing the query (except many-to-many relationships)
prefetch_related()	Select all related data when executing the query (including many-to-many relationships)
defer()	Do not retrieve the named fields from the database. Used to improve query performance on complex datasets
only()	Opposite of defer()—return only the named fields

using()	Select which database the QuerySet will be evaluated against (when using multiple databases)
select_for_update()	Return a QuerySet and lock table rows until the end of the transaction
raw()	Execute a raw SQL statement
AND (&)	Combine two QuerySets with the SQL AND operator. Using AND (&) is functionally equivalent to using filter() with multiple parameters
OR (|)	Combine two QuerySets with the SQL OR operator

FILTROS ORM

Ejemplo de consulta tipo 'filter'. Observe que el filtro deseado se adiciona al nombre de la columna uniendollo con los caracteres '__' (dos *underlines* seguidas)

```
xxx.objects.filter(firstname__startswith='L').values()
```

Keyword	Description
contains	**Contains the phrase**
icontains	**Same as contains, but case-insensitive**
date	**Matches a date**
day	**Matches a date (day of month, 1-31) (for dates)**
endswith	**Ends with**
iendswith	**Same as endswidth, but case-insensitive**
exact	**An exact match**
iexact	**Same as exact, but case-insensitive**
in	**Matches one of the values**

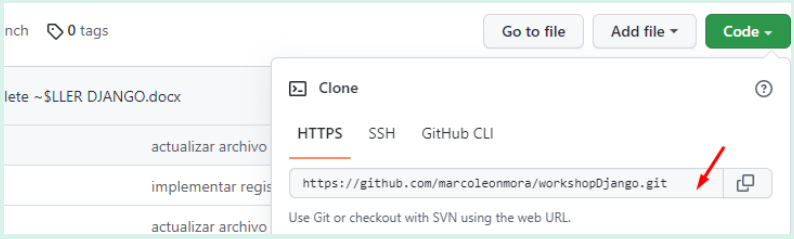
isnull	**Matches NULL values**
gt	**Greater than**
gte	**Greater than, or equal to**
hour	**Matches an hour (for datetimes)**
lt	**Less than**
lte	**Less than, or equal to**
minute	**Matches a minute (for datetimes)**
month	**Matches a month (for dates)**
quarter	**Matches a quarter of the year (1-4) (for dates)**
range	**Match between**
regex	**Matches a regular expression**
iregex	**Same as regex, but case-insensitive**
second	**Matches a second (for datetimes)**
startswith	**Starts with**
istartswith	**Same as startswith, but case-insensitive**
time	**Matches a time (for datetimes)**
week	**Matches a week number (1-53) (for dates)**
week_day	**Matches a day of week (1-7) 1 is Sunday**
iso_week_day	**Matches a ISO 8601 day of week (1-7) 1 is Monday**

year	**Matches a year (for dates)**
iso_year	**Matches an ISO 8601 year (for dates)**

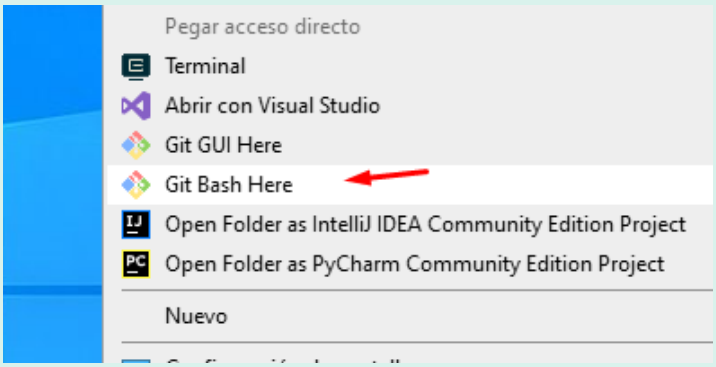
ANEXO 7. CONTROL DE VERSIONES CON GIT Y GITHUB

CLONAR UN REPOSITORIO

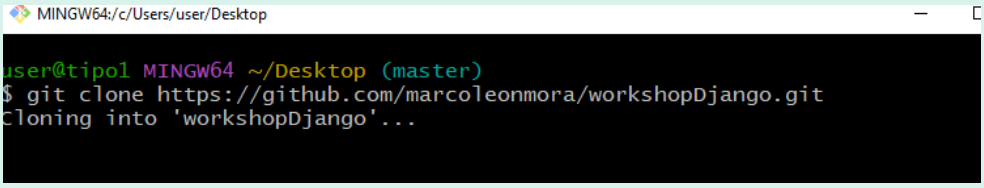
1. Se debe tener el link del repositorio a clonar



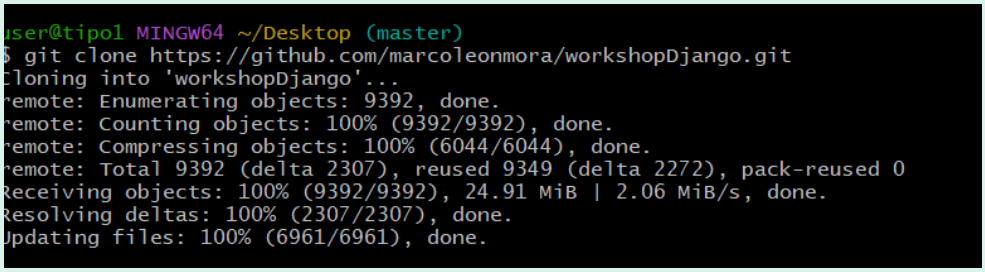
2. Abrir Git Bash en la ruta donde desea ubicar el proyecto



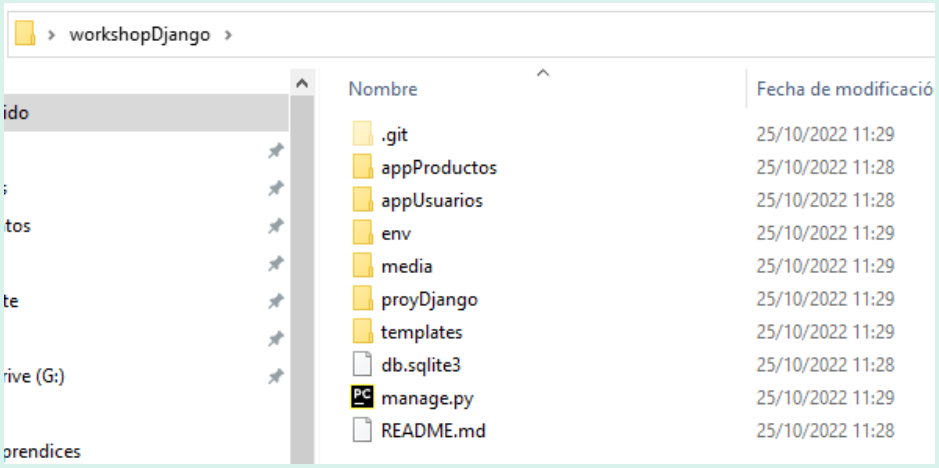
3. Digite el comando 'Git clone' y agregue la dirección URL que ha copiado antes.



4. Resultado



5. Los archivos se transfieren al escritorio el proyecto



SUBIR A REPOSITORIO REMOTO

6. Crear cuenta en github
7. Crear un repositorio en github
8. en la consola, activar el ambiente virtual: source env/Scripts/actívate
9. Iniciar el repositorio con el comando: git init
10. En directorio raíz crear un archivo: '.gitignore' (observe que inicia con un punto)
11. navegar al sitio: gitignore.io
12. En la casilla escribir: django y pulsar 'create'
13. Teclear <CTRL> + A, para seleccionar toda la página creada,
14. copiarla en el archivo '.gitignore' y guardar.
15. configurar la cuenta en el repositorio local:
    a. git config --global user.name "su nombre de usuario en github"
    b. git config --global user.email <su correo registrado en github>
16. agregar todos los archivos, con uno de los siguientes comandos (son equivalentes):
    a. git add -A
    b. git add .
17. Confirmar el primer estado (versión inicial): git commit -m "commit inicial"
18. Establecer la url del repositorio remoto (buscar la url en github): git remote add origin https://github.com/marcoleonmora/xxxxxxxxxxx.git
19. Cambiar a la rama main (en este caso la principal): git branch -M main
20. Subir los archivos al repositorio github: git push -u origin main

ENLACES

https://www.djangoproject.com/start/overview/

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/

https://openwebinars.net/blog/que-es-django-y-por-que-usarlo/

https://bedu.org/blog/tecnologia/3-razones-para-usar-el-framework-django#:~:text=Algunas%20de%20las%20caracter%C3%ADsticas%20de,manera%20estable%20y%20con%20rapidez.

https://www.freecodecamp.org/espanol/news/para-que-se-utiliza-django-de-python-5-razones-claves-por-las-que-uso-el-framework-django-para-proyectos/

https://umangsoftware.com/importance-of-ajax-in-web-development/

https://programarfacil.com/blog/que-es-un-orm/