

## Glosario

### ▼ Django

<b>Comando DJANGO-ADMIN:</b>	Comando para administrar un proyecto Django, como crear un proyecto o realizar migraciones.
<b>Comando StartProject:</b>	Comando para iniciar un proyecto Django.
<b>Configurar Settings.py:</b>	Proceso de configuración de archivos importantes como settings.py en un proyecto Django.
<b>Entorno Virtual:</b>	Un entorno aislado que permite gestionar las dependencias de un proyecto de forma independiente.
<b>Entorno de Desarrollo v/s Producción:</b>	Comparación entre los entornos de desarrollo y producción en un proyecto Django.
<b>Herencia de Componentes en el Modelo MVC:</b>	La herencia de componentes dentro de la estructura MVC para facilitar la reutilización de código en Django.
<b>Instalar Django:</b>	Proceso para instalar Django en un entorno virtual o global.
<b>Modelos en Django:</b>	Definición de la estructura de los datos en Django usando clases de Python.
<b>Servidor de Desarrollo:</b>	Servidor utilizado para probar el proyecto Django en el entorno de desarrollo.
<b>Templates en Django:</b>	Archivos HTML en Django que permiten mostrar contenido dinámico.
<b>Vistas en Django:</b>	Funciones o clases que reciben solicitudes y devuelven respuestas en Django.
<b>Vistas Heredadas:</b>	Reutilización de vistas de una aplicación en otras partes del proyecto Django.
<b>Widgets de Formularios en Django:</b>	Elementos utilizados para representar los campos de un formulario en las plantillas de Django.

### ▼ Modelos Auth de Django

<b>Accesos no Autorizados:</b>	Redirección de los usuarios que no están autorizados a acceder a ciertas páginas en Django.
<b>Autorización y Permisos:</b>	Modelo que maneja la autorización de los usuarios y sus permisos para acceder a ciertas partes del sistema.
<b>LoginRequiredMixin:</b>	Mixin utilizado para asegurar que un usuario esté autenticado antes de acceder a una vista.
<b>Mixins en Django:</b>	Clases que proporcionan funcionalidades adicionales que pueden ser utilizadas en vistas y otras partes del proyecto Django.
<b>Modelo Auth de Django:</b>	Modelo integrado en Django para gestionar la autenticación de usuarios y sus permisos.
<b>Permisos en Django:</b>	Definición de permisos específicos que los usuarios pueden tener sobre los objetos en un proyecto Django.

<b>Redirección de Accesos No Autorizados:</b>	Configuración para redirigir a los usuarios no autorizados a otra página, como una página de login.
<b>Vista de Autenticación:</b>	Vista que maneja el proceso de autenticación de usuarios en el proyecto Django.

#### ▼ Servidor de Administración de Django

<b>Accediendo al Sitio Administrativo:</b>	Proceso para acceder al panel de administración de Django, donde se gestionan los modelos y los usuarios.
<b>Administrador de Django:</b>	Herramienta de Django para administrar los modelos, usuarios y otros componentes del sistema a través de una interfaz web.
<b>Crear un Super Usuario:</b>	Comando para crear un superusuario que pueda acceder al panel de administración de Django.
<b>Limitar el Acceso al Sitio Administrativo:</b>	Restricción de acceso al sitio administrativo de Django para usuarios específicos.
<b>Manejo de Permisos en la Página Administrativa:</b>	Configuración y gestión de permisos para los usuarios que acceden al sitio administrativo de Django.

Navegue por el glosario usando este índice.

[Especial](#) | [A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [Ñ](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#) | [TODAS](#)

No se encontraron entradas en esta sección



## Contacta

 Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](#). Desarrollado por [Moodle](#)



## Aprendizaje esperado 1

### ¿Qué es Django?

- Django es un framework de desarrollo web basado en Python, diseñado para agilizar la creación de aplicaciones seguras y escalables.
- Su filosofía se basa en el principio **DRY** (Don't Repeat Yourself) y el enfoque de desarrollo rápido.

### ¿Por qué usar Django?

- **Rápido y eficiente:** Reduce el tiempo de desarrollo con herramientas integradas.
- **Seguro:** Incluye protección contra ataques como SQL Injection y Cross-Site Scripting.
- **Escalable:** Permite construir aplicaciones que crecen sin problemas.
- **Versátil:** Útil para aplicaciones empresariales, API REST, sistemas de administración, entre otros.

### Flexibilidad de Instalación y Configuración

- Se puede instalar en múltiples plataformas (Windows, Linux, macOS).
- Soporta configuración personalizada mediante el archivo `settings.py`.

### El Entorno de Desarrollo Django

- Django incluye un servidor de desarrollo para pruebas rápidas.
- Permite depuración eficiente mediante `DEBUG = True` en `settings.py`.

### Entorno de Desarrollo vs Producción

- **Desarrollo:** Uso del servidor integrado de Django, depuración activa y recarga automática.
- **Producción:** Uso de servidores como Gunicorn o uWSGI con Nginx y configuración optimizada de seguridad.

### Django vs Python

- Django es un framework construido en Python, pero añade herramientas específicas para desarrollo web.
- Permite aplicar las buenas prácticas y la sintaxis limpia de Python a proyectos web.

### Django y la Estructura Web para el Desarrollo

- Django sigue la arquitectura **MVC (Modelo-Vista-Controlador)** adaptada como **MTV (Modelo-Template-Vista)**.
- Separa la lógica de negocio (Modelos), la presentación (Templates) y la gestión de datos (Vistas).

### Soporte para Bases de Datos

- Soporta varias bases de datos como PostgreSQL, MySQL, SQLite y Oracle.
- Usa el **ORM (Object-Relational Mapper)** para interactuar con bases de datos sin escribir SQL directamente.

### Python y los Entornos Virtuales

- Un **entorno virtual** es un espacio aislado donde se instalan dependencias sin afectar el sistema global.
- Se recomienda usar entornos virtuales para cada proyecto.

### ¿Por qué usar entornos virtuales?

- **Aislamiento:** Evita conflictos de dependencias entre proyectos.
- **Gestión de versiones:** Permite usar diferentes versiones de Django en distintos proyectos.

- **Facilidad de instalación:** Mejora la organización y mantenimiento de paquetes.

### El Entorno Virtual: Velocidad de Desarrollo

- Mantiene un entorno limpio con las librerías necesarias.
- Facilita el despliegue en servidores con configuraciones controladas.

### Estructura Minimalista

- Django sigue un diseño modular con aplicaciones independientes.
- Usa un patrón claro de directorios: `models.py`, `views.py`, `templates/`, `static/`, etc.

### Estructura Flexible

- Permite extender y modificar componentes según las necesidades del proyecto.
- Compatible con múltiples herramientas externas.

### Librerías Propias de Cada Proyecto

- Cada entorno virtual puede contener librerías específicas sin afectar otros proyectos.
- Uso del archivo `requirements.txt` para gestionar dependencias.

### Aislamiento del Entorno Python

- Mantiene paquetes y configuraciones dentro del entorno virtual.
- Evita interferencias con otros proyectos o aplicaciones del sistema.

### Manejo de Distintas Versiones

- Permite instalar versiones específicas de Django para cada proyecto.
- Evita incompatibilidades entre versiones antiguas y nuevas.

### El Comando venv

- Se usa para crear entornos virtuales en Python:

### Iniciando el Entorno Virtual

- Activar en Windows:  
`mi_entorno\Scripts\activate`
- Activar en macOS/Linux:  
`source mi_entorno/bin/activate`

### Saliendo del Entorno Virtual

- Para salir del entorno virtual:  
`deactivate`

### Instalaciones en el Entorno Global

- Evitar instalar paquetes directamente en el sistema operativo.
- Siempre trabajar con entornos virtuales para mejor administración.

### El Enrutador de Django

- Django usa un sistema de enrutamiento basado en `urls.py`.

### MVC en Django para Aplicaciones Monolíticas

- Django sigue el patrón **MTV (Modelo-Template-Vista)**:
  - **Modelo (`models.py`)**: Maneja la base de datos.
  - **Vista (`views.py`)**: Contiene la lógica de negocio.



- **Template (templates/)**: Define la interfaz de usuario.

#### **Herencia de Componentes en el Modelo MVC**

- Permite reutilizar código en los templates con herencia.

#### **El Principio DRY y su Aplicación en el Entorno Python/Django**

- **DRY (Don't Repeat Yourself)**: Reutilización de código para evitar redundancia.
- Ejemplo: Uso de formularios genéricos en Django.

#### **¿Qué son los Templates de Django?**

- Son archivos HTML con etiquetas de Django para renderizar contenido dinámico.
- Permiten el uso de variables y lógica sencilla dentro de la vista.

#### **Renderización en Django**

- Django usa la función `render()` para enviar datos a los templates.

Última modificación: viernes, 2 de mayo de 2025, 19:52



## **Contacta**

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](http://LMSACE.com). Desarrollado por [Moodle](http://Moodle)



## Introducción a Django

### Objetivos

- Comprender qué es Django y cuáles son sus ventajas en el desarrollo web.
- Instalar y configurar Django en un entorno de desarrollo.
- Diferenciar entre el entorno de desarrollo y producción en Django.
- Entender la relación entre Django y Python.
- Conocer la estructura de un proyecto Django y su organización.
- Identificar los distintos sistemas de bases de datos que Django soporta y cómo configurarlos.

### ¿Qué es Django?

Django es un framework de desarrollo web de alto nivel escrito en Python, diseñado para facilitar la creación rápida de aplicaciones web seguras y escalables. Se basa en el principio "**"Don't Repeat Yourself" (DRY)**", promoviendo la reutilización del código y la eficiencia en el desarrollo.

Django sigue el patrón **Modelo-Vista-Template (MVT)**, proporcionando herramientas integradas para la gestión de bases de datos, autenticación de usuarios, administración de contenido y más.

### ¿Por qué usar Django?

Django es una excelente opción para el desarrollo web por varias razones:

- **Rapidez de desarrollo:** Incluye herramientas y funcionalidades listas para usar, reduciendo el tiempo de desarrollo.
- **Seguridad:** Protege contra amenazas comunes como inyección SQL, Cross-Site Scripting (XSS) y Cross-Site Request Forgery (CSRF).
- **Escalabilidad:** Puede manejar aplicaciones desde pequeños proyectos hasta plataformas de alto tráfico.
- **Comunidad activa:** Cuenta con una gran cantidad de documentación y soporte de desarrolladores en todo el mundo.
- **Compatibilidad con bases de datos:** Soporta múltiples bases de datos como PostgreSQL, MySQL, SQLite y Oracle.

### Flexibilidad de instalación y configuración

Django se puede instalar fácilmente mediante `pip`, el gestor de paquetes de Python. Su configuración es flexible, permitiendo ajustes personalizados en su archivo `settings.py`.

Django también permite la configuración de entornos personalizados para **desarrollo** y **producción**, lo que facilita su adaptación a distintas necesidades.

### El entorno de desarrollo Django

El entorno de desarrollo de Django ofrece herramientas para agilizar la construcción y prueba de aplicaciones:

- **Servidor de desarrollo integrado:** Permite probar aplicaciones sin necesidad de configurar un servidor externo.
- **Consola interactiva:** Facilita la ejecución de comandos y pruebas en Python.
- **Sistema de migraciones:** Permite gestionar cambios en la base de datos de forma sencilla.

### Entorno de desarrollo vs producción



Característica	Desarrollo	Producción
Modo de depuración	Activo ( <code>DEBUG=True</code> )	Desactivado ( <code>DEBUG=False</code> )
Base de datos	SQLite (por defecto)	PostgreSQL, MySQL u Oracle
Seguridad	Mínima (para pruebas)	Máxima (protección contra ataques)

Servidor web	Integrado en Django	dedicado (Gunicorn, Nginx, Apache)
--------------	---------------------	------------------------------------

En producción, se recomienda utilizar configuraciones avanzadas de seguridad y escalabilidad, como el uso de bases de datos robustas y servidores optimizados.

## Django vs Python

Aunque Django está escrito en Python, hay diferencias clave entre el lenguaje y el framework:

- **Python** es un lenguaje de programación versátil que se usa en diversos ámbitos (ciencia de datos, automatización, IA, etc.).
- **Django** es un framework web basado en Python que proporciona herramientas específicas para el desarrollo de aplicaciones web.

Django no reemplaza a Python, sino que extiende sus capacidades para el desarrollo web.

## Django y la estructura web para el desarrollo

Django organiza los proyectos en una estructura modular:

```
mi_proyecto/
|__ mi_proyecto/
|   |__ settings.py      # Configuración del proyecto
|   |__ urls.py          # Rutas de la aplicación
|   |__ wsgi.py           # Configuración para servidores web
|
|__ mi_app/
|   |__ models.py        # Definición de la base de datos
|   |__ views.py         # Lógica de negocio
|   |__ templates/       # Archivos HTML para la interfaz
|
|__ manage.py           # Herramienta de administración de Django
```

Esta estructura facilita la organización del código y la separación de responsabilidades.

## Soporte para bases de datos

Django incluye un ORM (Object-Relational Mapping) que permite interactuar con bases de datos sin escribir SQL directamente. Soporta:

- **PostgreSQL** (recomendada para producción)
- **MySQL**
- **SQLite** (por defecto en desarrollo)
- **Oracle**

Última modificación: viernes, 2 de mayo de 2025, 19:52



## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](#). Desarrollado por [Moodle](#)

^

?

## Python y los entornos virtuales

### Objetivos

- Comprender qué es un entorno virtual en Python y su propósito.
- Crear y gestionar entornos virtuales utilizando `venv` y `virtualenv`.
- Identificar las ventajas de los entornos virtuales en la velocidad de desarrollo.
- Diferenciar entre una estructura minimalista y una estructura flexible en los entornos virtuales.
- Entender la importancia del aislamiento de entornos y el manejo de librerías específicas por proyecto.
- Gestionar distintas versiones de Python en entornos virtuales.

### Python y los entornos virtuales

Un **entorno virtual en Python** es un espacio aislado donde se pueden instalar librerías y dependencias sin afectar la configuración global del sistema.

Esto permite:

- Evitar conflictos entre proyectos con distintas versiones de librerías.
- Mantener cada aplicación con su propio conjunto de dependencias.
- Probar y desarrollar software en entornos controlados.

Los entornos virtuales son esenciales en el desarrollo de aplicaciones Python, especialmente en proyectos colaborativos o que requieren despliegue en servidores.

### ¿Por qué usar entornos virtuales?

Sin entornos virtuales, todas las librerías se instalan en el sistema global de Python. Esto puede causar problemas como:

- **Conflictos de versiones:** Un proyecto puede necesitar una versión específica de una librería, mientras otro requiere una diferente.
- **Dificultades en la reproducción del entorno:** Sin un entorno virtual, compartir un proyecto con otros desarrolladores puede ser complicado.
- **Dependencias innecesarias:** Instalar librerías a nivel global puede llenar el sistema de paquetes innecesarios.

Un entorno virtual soluciona estos problemas al proporcionar un espacio aislado para cada proyecto.

### El entorno virtual: velocidad de desarrollo

Trabajar con entornos virtuales mejora la productividad porque:

- Permite instalar dependencias específicas para cada proyecto sin interferencias.
- Asegura que los proyectos sean reproducibles en distintos entornos.
- Facilita la instalación y actualización de paquetes sin afectar otros proyectos.

### Estructura minimalista

Un entorno virtual tiene una estructura simple y ligera. Al crearlo, solo incluye los archivos necesarios para ejecutar Python y administrar paquetes.



```
mi_entorno/
|--- bin/ (ejecutables en Linux/macOS)
|--- Scripts/ (ejecutables en Windows)
|--- include/ (archivos de cabecera de C si se usan)
|--- lib/ (librerías instaladas)
|--- pyvenv.cfg (configuración del entorno virtual)
```

Este diseño evita la sobrecarga de dependencias innecesarias.

### Estructura flexible

Aunque minimalista, un entorno virtual permite instalar cualquier paquete requerido para el desarrollo del proyecto.

Cada entorno virtual mantiene sus propias versiones de librerías sin afectar el sistema global.

### Librerías propias de cada proyecto

Dentro de un entorno virtual, solo se instalan las librerías necesarias para el proyecto. Esto garantiza que el entorno sea ligero y libre de dependencias innecesarias.

### Aislamiento del entorno Python

El aislamiento es una de las principales ventajas de los entornos virtuales:

- Cada entorno tiene su propia instalación de Python y paquetes.
- No interfiere con otras aplicaciones en el sistema.
- Permite probar nuevas versiones de librerías sin afectar otros proyectos.

Este aislamiento evita conflictos y hace que los proyectos sean más estables y predecibles.

### Manejo de distintas versiones

Los entornos virtuales también permiten trabajar con diferentes versiones de Python en un mismo sistema.

Para crear un entorno virtual con una versión específica de Python:

```
python3.9 -m venv mi_entorno_39
python3.11 -m venv mi_entorno_311
```

Verificación de la versión de Python dentro del entorno:

```
python --version
```

Herramientas como **pyenv** permiten instalar y administrar múltiples versiones de Python en el sistema, facilitando el uso de entornos virtuales con diferentes versiones.

Los entornos virtuales en Python son una herramienta fundamental para el desarrollo profesional. Proporcionan aislamiento, flexibilidad y control sobre las dependencias de cada proyecto, garantizando un flujo de trabajo eficiente y ordenado.

Última modificación: viernes, 2 de mayo de 2025, 19:53

## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](#). Desarrollado por [Moodle](#)

^

?

## Uso de entornos virtuales y arquitectura en Django

### Objetivos

- Comprender y utilizar el comando `venv` para crear entornos virtuales en Python.
- Activar y desactivar entornos virtuales en distintos sistemas operativos.
- Distinguir entre instalaciones en el entorno global y en un entorno virtual.
- Entender el papel del enrutador en Django y su relación con las aplicaciones web.
- Aplicar el modelo MVC en Django para el desarrollo de aplicaciones monolíticas.
- Implementar herencia de componentes en el modelo MVC.
- Comprender el principio DRY y su aplicación en el desarrollo con Python y Django.
- Utilizar templates en Django para estructurar la presentación de datos.
- Renderizar contenido dinámico en Django mediante templates y vistas.

### El comando `venv`

El comando `venv` permite crear entornos virtuales en Python, aislando las dependencias de cada proyecto.

#### Creación de un entorno virtual

Para crear un entorno virtual, se ejecuta:

```
python -m venv mi_entorno
```

Esto genera una carpeta con la estructura del entorno virtual, incluyendo los ejecutables de Python y un espacio para instalar paquetes específicos del proyecto.

#### Iniciando el entorno virtual

Para activar un entorno virtual en diferentes sistemas operativos:

```
# En macOS y Linux  
source mi_entorno/bin/activate  
  
# En Windows  
mi_entornoScriptsactivate
```

Una vez activado, cualquier instalación de paquetes con `pip` se almacenará en el entorno virtual y no afectará el sistema global.

#### Saliendo del entorno virtual

Para salir del entorno virtual y volver al entorno global del sistema, se usa el comando:

```
deactivate
```

Esto restaura la configuración predeterminada de Python y evita conflictos con otros proyectos.

#### Instalaciones en el entorno global

Si se instalan paquetes sin un entorno virtual activado, estos quedan disponibles para todo el sistema.



```
pip install django
```

sto puede generar conflictos si distintas versiones de Django son necesarias en distintos proyectos. Por eso, es recomendable usar entornos virtuales.

## El enrutador de Django

Django maneja las solicitudes de los usuarios mediante un **enrutador**, que determina qué vista ejecutar según la URL solicitada.

### Definición de rutas

Las rutas se definen en `urls.py` dentro de la aplicación Django.

Cuando un usuario accede a `https://midominio.com/contacto/`, Django ejecuta la vista `contacto`.

## MVC en Django para aplicaciones monolíticas

Django sigue el patrón **Modelo-Vista-Controlador (MVC)**, aunque con una terminología diferente:

- **Modelo (Model)**: Representa los datos y la lógica de negocio.
- **Vista (View)**: Maneja la lógica de presentación y respuesta.
- **Controlador (Controller)**: En Django, la funcionalidad del controlador se integra en las vistas y en el sistema de enrutamiento.

## Herencia de componentes en el modelo MVC

Django permite reutilizar código en las vistas y plantillas mediante **herencia de componentes**.

## El principio DRY y su aplicación en el entorno Python/Django

**DRY (Don't Repeat Yourself)** es un principio que promueve evitar la duplicación de código.

En Django, DRY se aplica mediante:

- **Herencia de templates**, como en el ejemplo anterior.
- **Uso de funciones y clases reutilizables** en las vistas.
- **Definición de modelos reutilizables** para evitar redundancia en la base de datos.

## Qué son los templates de Django

Los **templates en Django** son archivos HTML que pueden contener **sintaxis dinámica** para mostrar datos.

Ejemplo de template con datos dinámicos:

```
<h2>Lista de Productos</h2>
<ul>
    {% for producto in productos %}
        <li>{{ producto.nombre }} - ${{ producto.precio }}</li>
    {% endfor %}
</ul>
```

Django reemplaza `{{ producto.nombre }}` y `${{ producto.precio }}` con datos reales desde la base de datos.

## Renderización en Django

El motor de templates de Django renderiza HTML dinámicamente usando los datos proporcionados en las vistas.

Última modificación: viernes, 2 de mayo de 2025, 19:53



## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](#). Desarrollado por [Moodle](#)

^

?

[Ver](#) [Hacer un envío](#)

## Ejercicio individual

### Objetivo

Comprender los conceptos teóricos fundamentales de Django, su estructura, entorno de desarrollo y la importancia de los entornos virtuales.

### Instrucciones

#### 1. Investigación Guiada:

- Investiga y responde las siguientes preguntas en tus propias palabras:
  - ¿Qué es Django y por qué se usa?
  - Diferencias entre el entorno de desarrollo y producción en Django.
  - Comparación entre Django y Python: ¿cómo se relacionan?
  - ¿Por qué Django facilita el desarrollo de aplicaciones web?
  - ¿Qué bases de datos soporta Django?
  - ¿Qué es un entorno virtual en Python y por qué es útil?
  - ¿Cómo se crea y se usa un entorno virtual en Python?

#### 2. Análisis de la Estructura de Django:

- Explica en un párrafo la estructura general de un proyecto Django y su enfoque MVC.
- Describe cómo Django maneja el enrutamiento de URLs.

#### 3. Reflexión sobre las Buenas Prácticas:

- Explica el principio **DRY (Don't Repeat Yourself)** y cómo se aplica en Django.
- ¿Qué ventajas tiene la herencia de componentes en Django?

#### 4. Exploración de los Templates en Django:

- Define qué son los templates en Django y su función en el desarrollo web.
- Explica cómo Django renderiza las vistas con templates.

### Entrega

- Documento Word o PDF.
- Ejecución: Individual
- Duración: 1 jornada de clases

## Sumario de calificaciones

Ocultado a los estudiantes	No
Participantes	34
Enviados	0
Pendientes por calificar	0

?



## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](#). Desarrollado por [Moodle](#)

^

?

## Aprendizaje esperado 2

**Utilizar las herramientas administrativas provistas por el framework para la configuración de un nuevo proyecto web Django**

### Instalando Django: El utilitario pip

- Descripción del proceso de instalación utilizando el gestor de paquetes pip.
- Importancia de realizar la instalación en un entorno controlado.

### Instalación en un entorno virtual

- Beneficios de trabajar con entornos virtuales en el desarrollo web con Django.
- Procedimientos para crear y activar un entorno virtual para un proyecto Django.

### Comprobar la instalación de Django

- Comandos y métodos para verificar si Django está correctamente instalado en el entorno virtual.
- Verificación de la versión de Django instalada.

### Creando un proyecto Django: Instrucción para crear un proyecto

- Proceso para crear un nuevo proyecto Django desde cero.
- Descripción de los pasos a seguir al utilizar el comando adecuado para iniciar un proyecto.

### El comando django-admin

- Descripción del comando `django-admin` y su utilidad en la administración del proyecto.
- Diferencia entre `django-admin` y `manage.py`.

### El comando startproject

- Explicación del comando `startproject` para iniciar la estructura básica de un proyecto Django.
- Entender cómo se organiza la estructura de carpetas y archivos.

### Estructura de carpetas de un proyecto Django

- Detalle de la estructura generada por el comando `startproject`.
- Explicación del propósito de cada archivo y carpeta creada en el proyecto.

### El utilitario manage.py: runserver, startup, createsuperuser, otros comandos

- Uso de `manage.py` para ejecutar diversos comandos administrativos, tales como `runserver` y `createsuperuser`.
- Explicación del flujo de trabajo básico y cómo interactuar con el servidor de desarrollo de Django.

### Archivos de configuración: `__init__.py`, `settings.py`

- Explicación del archivo `__init__.py` y su rol en la organización del proyecto.
- Descripción del archivo `settings.py` como el núcleo de la configuración del proyecto.

### Manejo de espacios de nombre



- Importancia de los espacios de nombre en Django para estructurar y organizar el código del proyecto de manera eficiente.

### Levantando el servidor con manage.py



- Procedimiento para iniciar el servidor de desarrollo de Django y cómo interactuar con el proyecto en el navegador.

## Revisando el proyecto web de Django

- Métodos para revisar y acceder a la página web creada con Django en el entorno de desarrollo local.

## Creando aplicaciones en Django, MVC y la creación de aplicaciones

- [Introducción](#) al patrón de arquitectura MVC en Django y su aplicación en el desarrollo de proyectos web.
- Cómo crear y agregar aplicaciones al proyecto Django.

## Correlación del modelo

- Explicación de cómo el modelo se integra con las vistas y controladores en Django, en el contexto de MVC.

## Las componentes esenciales

- Componentes clave en Django: modelos, vistas, plantillas (templates), y URL routing.

## Comandos de ayuda Django

- Descripción de los comandos de ayuda de Django para obtener asistencia al usar `django-admin` y `manage.py`.

## Configurando un proyecto: Configurando settings.py

- Explicación detallada de cómo configurar `settings.py` para adaptar el proyecto a las necesidades específicas del desarrollo.

## Configuración de templates

- Configuración de la carpeta de plantillas en Django para facilitar la creación de vistas dinámicas.

## Configuración de paths

- Descripción de cómo gestionar los paths de archivos y directorios dentro del proyecto Django.

## Configuración urls.py

- Definición de las URL en Django y cómo configurarlas para enlazar vistas y aplicaciones correctamente.

Última modificación: viernes, 2 de mayo de 2025, 19:54



## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](#). Desarrollado por [Moodle](#)



## Instalación de Django en un entorno virtual

### Objetivos

- Comprender el uso de `pip` para instalar paquetes en Python, específicamente Django.
- Instalar Django en un entorno virtual para aislar las dependencias del proyecto.
- Verificar la instalación de Django y asegurarse de que todo esté configurado correctamente.

### Instalando Django: El utilitario `pip`

`pip` es el administrador de paquetes de Python y es la herramienta principal para instalar y gestionar bibliotecas y dependencias en proyectos de Python, incluyendo Django.

#### ¿Qué es `pip`?

`pip` es una herramienta que se utiliza para instalar y administrar paquetes Python que se encuentran en el repositorio oficial de Python (PyPI). Es indispensable para cualquier desarrollador de Python que desee integrar librerías externas en sus proyectos.

#### Instalación de `pip`

En la mayoría de las instalaciones modernas de Python, `pip` viene preinstalado. Sin embargo, si no está disponible, puedes instalarlo manualmente. Para ello, ejecuta el siguiente comando en la terminal:

```
python -m ensurepip --upgrade
```

### Instalación en un entorno virtual

Antes de instalar Django, es recomendable trabajar dentro de un **entorno virtual** para asegurarnos de que las dependencias de diferentes proyectos no interfieran entre sí. Un entorno virtual es una copia aislada de Python donde podemos instalar paquetes sin afectar el sistema global.

#### Crear un entorno virtual

Primero, creamos un entorno virtual utilizando el comando `venv`. Esto nos permitirá trabajar de manera aislada:

```
python -m venv mi_entorno
```

Este comando creará una carpeta llamada `mi_entorno` que contendrá una instalación independiente de Python y `pip` para gestionar las dependencias.

#### Activar el entorno virtual

Para activar el entorno virtual, usa el siguiente comando según tu sistema operativo:

```
# En macOS y Linux  
source mi_entorno/bin/activate  
  
# En Windows  
mi_entorno\Scripts\activate
```

Verás que el nombre del entorno aparece en la línea de comandos, indicando que estás trabajando dentro de este entorno aislado.

#### Instalar Django en el entorno virtual

Una vez activado el entorno virtual, puedes proceder a instalar Django. Asegúrate de que el entorno virtual esté activado

?

antes de ejecutar el siguiente comando:

```
pip install django
```

Este comando descargará e instalará la última versión de Django desde PyPI dentro del entorno virtual.

### Verificar la instalación de Django

Para comprobar que Django se ha instalado correctamente, puedes ejecutar el siguiente comando en la terminal:

```
django admin --version
```

Si todo está bien, este comando debería mostrar la versión de Django instalada. Algo similar a: **4.1.2**

### Comprobar la instalación de Django

Para verificar que Django está correctamente instalado, además de revisar la versión como se explicó previamente, podemos realizar algunos pasos adicionales:

#### Crear un proyecto de prueba

Django incluye una herramienta de línea de comandos llamada `django-admin` que facilita la creación y gestión de proyectos y aplicaciones. Después de instalar Django, puedes probar creando un proyecto básico:

1. Ejecuta el siguiente comando para crear un proyecto nuevo:

```
django admin startproject mi_sitio
```

2. Esto creará una estructura de proyecto básica con los archivos y directorios necesarios.

3. Luego navega a la carpeta del proyecto:

```
cd mi_sitio
```

4. Finalmente, ejecuta el servidor de desarrollo de Django para comprobar que todo esté funcionando correctamente:

```
phyton manage.py runserver
```

Este comando debería incluir el servidor de desarrollo y mostrar un mensaje como el siguiente:

```
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

5. Abre tu navegador y accede a <http://127.0.0.1:8000/>. Si ves la página predeterminada de Django, entonces la instalación fue exitosa.

Última modificación: viernes, 2 de mayo de 2025, 19:54



## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)



Copyright © 2017 -Desarrollado por [LMSACE.com](http://LMSACE.com). Desarrollado por [Moodle](http://Moodle)

?

## Creando un proyecto de Django

### Objetivos

- Crear un proyecto Django desde cero utilizando el comando `django-admin` y `startproject`.
- Conocer la estructura de carpetas generada automáticamente por Django.
- Comprender el funcionamiento del utilitario `manage.py` y sus comandos principales.
- Familiarizarse con los archivos de configuración esenciales en un proyecto Django.
- Levantar el servidor de desarrollo y revisar el proyecto web generado.

### Creando un proyecto Django: Instrucción para crear un proyecto

Django facilita la creación de un proyecto web completo con una estructura básica y funcional. Para iniciar un nuevo proyecto Django, se utiliza el comando `django-admin` con el subcomando `startproject`. Este comando crea un conjunto de archivos y carpetas que forman la base de tu aplicación web.

#### Comando `django-admin`

El comando `django-admin` es una herramienta de línea de comandos que facilita la administración de proyectos y aplicaciones Django. Se utiliza para realizar diversas tareas como la creación de un nuevo proyecto, la ejecución de migraciones, la creación de superusuarios, entre otros.

#### Comando `startproject`

El comando `startproject` es el primero que se usa cuando se quiere iniciar un proyecto Django. Este comando crea un nuevo directorio con el nombre que se le asigne y coloca dentro una estructura básica de archivos y carpetas necesarias para el proyecto.

```
django-admin startproject mi_proyecto
```

Este comando crea una carpeta llamada `mi_proyecto`, y dentro de ella se encuentran los archivos esenciales para un proyecto Django.

#### Estructura de carpetas de un proyecto Django

Cuando ejecutas el comando `startproject`, Django crea automáticamente una serie de archivos y carpetas. La estructura básica es la siguiente:

```
mi_proyecto/
  manage.py
  mi_proyecto/
    __init__.py
    settings.py
    urls.py
    asgi.py
    wsgi.py
```

#### Descripción de las carpetas y archivos generados

- **`manage.py`**: Es el script de administración principal. Permite ejecutar comandos de gestión como iniciar el servidor, aplicar migraciones, y crear superusuarios.
- **`mi_proyecto/`**: Es la carpeta que contiene los archivos de configuración específicos de tu proyecto.
  - **`__init__.py`**: Un archivo vacío que indica que la carpeta debe ser tratada como un paquete de Python.
  - **`settings.py`**: Contiene la configuración principal del proyecto, como la configuración de la base de datos, rutas,

middleware, entre otros.

- **urls.py**: Archivo que se usa para definir las rutas o URLs de la aplicación.
- **asgi.py**: Archivo de configuración para ASGI (Asynchronous Server Gateway Interface), utilizado para aplicaciones asíncronas.
- **wsgi.py**: Archivo de configuración para WSGI (Web Server Gateway Interface), utilizado para aplicaciones síncronas.

### El utilitario `manage.py`: runserver, startup, createsuperuser, otros comandos

El archivo `manage.py` es una herramienta crucial para interactuar con Django desde la terminal. A continuación, describimos algunos de los comandos más comunes que puedes utilizar con `manage.py`.

#### `runserver`

El comando `runserver` levanta un servidor web de desarrollo para tu proyecto Django. Es útil para probar y desarrollar la aplicación localmente. Puedes ejecutar el siguiente comando:

```
python manage.py runserver
```

Esto iniciará el servidor en `http://127.0.0.1:8000/` por defecto. Puedes acceder a esta dirección desde tu navegador para ver el proyecto Django funcionando.

#### `startapp`

Este comando se usa para crear una nueva aplicación dentro del proyecto Django. Una aplicación es una parte de la funcionalidad de tu sitio web, como un sistema de blog, una tienda en línea, etc.

```
python manage.py startapp nombre_de_la_app
```

#### `createsuperuser`

Este comando se usa para crear un superusuario, el cual tiene acceso a todas las funcionalidades del sitio de administración de Django. Para ejecutarlo, utiliza:

```
python manage.py createsuperuser
```

Te pedirá ingresar un nombre de usuario, correo electrónico y contraseña para el superusuario.

#### Otros comandos

- **makemigrations**: Crea las migraciones de la base de datos.
- **migrate**: Aplica las migraciones y actualiza la base de datos.
- **shell**: Abre una shell interactiva de Python donde puedes interactuar con el proyecto Django.
- **test**: Ejecuta las pruebas definidas en el proyecto.

### Archivos de configuración: `__init__.py`, `settings.py`

#### `__init__.py`

Este archivo vacío se encuentra dentro de la carpeta del proyecto y su único propósito es indicar que la carpeta debe ser tratada como un módulo de Python.

#### `settings.py`

El archivo `settings.py` es el corazón de la configuración de tu proyecto Django. Aquí se definen parámetros como:

- **INSTALLED\_APPS**: Una lista de aplicaciones que el proyecto usará.
- **DATABASES**: Configuración de la base de datos (por defecto usa SQLite).
- **MIDDLEWARE**: Configuración de middleware que procesará las solicitudes.

- **TEMPLATES**: Configuración de plantillas HTML.

- **STATIC\_URL**: Ruta de los archivos estáticos.

Es importante modificar este archivo según las necesidades de tu proyecto, como cambiar la base de datos, habilitar aplicaciones adicionales y configurar rutas.

### Manejo de espacios de nombre

En Django, el manejo de espacios de nombre se refiere a cómo se organizan las rutas y las aplicaciones dentro del proyecto. Los espacios de nombre ayudan a evitar colisiones de nombres en aplicaciones de Django que puedan tener rutas o funciones similares.

Por ejemplo, en el archivo `urls.py` de cada aplicación, puedes definir un espacio de nombre para tus rutas:

```
app_name = 'mi_aplicacion'

urlpatterns = [
    path('', views.index, name='index'),
]
```

Esto permite referenciar las rutas dentro del proyecto de forma única utilizando el nombre del espacio de nombre.

### Levantando el servidor con `manage.py`

Para iniciar el servidor de desarrollo de Django, se utiliza el siguiente comando:

```
python manage.py runserver
```

### Revisando el proyecto web de Django

Una vez que hayas ejecutado el servidor con `runserver`, puedes acceder a tu proyecto en tu navegador y deberías ver la página predeterminada de Django. Esto indica que la instalación y configuración básica del proyecto se han realizado correctamente.

Última modificación: viernes, 2 de mayo de 2025, 19:54



## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](http://LMSACE.com). Desarrollado por [Moodle](http://Moodle)



## Django y el modelo MVC

### Objetivos

- Comprender la estructura MVC (Modelo-Vista-Controlador) en Django.
- Crear aplicaciones dentro de un proyecto Django.
- Configurar correctamente `settings.py` para incluir nuevas aplicaciones.
- Entender la correlación entre modelos, vistas y controladores en Django.
- Utilizar los comandos de ayuda de Django para administrar el desarrollo.
- Configurar `templates`, `paths` y `urls.py` para definir rutas y vistas en Django.

### Creando Aplicaciones en Django y el Modelo MVC

Django sigue un patrón de diseño similar a MVC (Modelo-Vista-Controlador), aunque con una nomenclatura propia conocida como MVT (Modelo-Vista-Template). Este enfoque permite separar la lógica de negocio, la presentación y la manipulación de datos en diferentes capas.

### Estructura MVC en Django (MVT en Django)

Componente	En Django
<b>Modelo (Model)</b>	Representa la estructura de los datos y se define en <code>models.py</code> .
<b>Vista (View)</b>	Contiene la lógica de presentación y se define en <code>views.py</code> .
<b>Controlador (Controller)</b>	Es manejado por el enrutador de Django y las URLs en <code>urls.py</code> .

En Django, el "Template" reemplaza la capa de "Vista" tradicional, y las vistas (`views.py`) actúan como controladores.

### Creando una Aplicación en Django

En Django, una aplicación es un módulo que encapsula una funcionalidad específica dentro del proyecto. Para crear una nueva aplicación dentro de un proyecto Django, se usa el siguiente comando:

```
python manage.py startapp nombre_de_la_app
```

Este comando generará la siguiente estructura:

```
nombre_de_la_app/
  migrations/
    __init__.py
  admin.py
  apps.py
  models.py
  tests.py
  views.py
```

- **models.py**: Define los modelos de datos (tablas en la base de datos).
- **views.py**: Contiene la lógica para manejar las solicitudes y generar respuestas.
- **admin.py**: Configuración del panel de administración de Django.
- **urls.py** (se debe crear manualmente en la aplicación): Define las rutas de la aplicación.
- **migrations/**: Contiene las migraciones de la base de datos.

### Correlación del Modelo en Django

En Django, los modelos representan las estructuras de datos en la base de datos utilizando clases de Python. Cada clase en

`models.py` se traduce en una tabla dentro de la base de datos.

Ejemplo de un modelo en Django:

```
from django.db import models

class Cliente(models.Model):
    nombre = models.CharField(max_length=100)
    email = models.EmailField(unique=True)
    fecha_registro = models.DateTimeField(auto_now_add=True)
```

Para reflejar este modelo en la base de datos, se ejecutan los siguientes comandos:

```
python manage.py makemigrations
python manage.py migrate
```

### Los Componentes Esenciales de Django

Para que una aplicación Django funcione correctamente, es necesario configurar ciertos archivos esenciales dentro del proyecto.

#### Configurando `settings.py` para incluir una aplicación

Una vez creada la aplicación, es necesario registrarla en `settings.py` dentro de la lista `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'nombre_de_la_app', # Se añade la nueva aplicación aquí
]
```

### Comandos de Ayuda en Django

Django proporciona una serie de comandos útiles para el desarrollo y la administración del proyecto.

Comando	Descripción
<code>python manage.py startapp nombre_de_la_app</code>	Crea una nueva aplicación
<code>python manage.py runserver</code>	Inicia el servidor de desarrollo
<code>python manage.py makemigrations</code>	Crea nuevas migraciones para la base de datos
<code>python manage.py migrate</code>	Aplica las migraciones en la base de datos
<code>python manage.py createsuperuser</code>	Crea un usuario administrador para el panel de Django.
<code>python manage.py shell</code>	Abre una shell interactiva con acceso a Django.

### Configuración de Templates en Django

Django usa templates para manejar la capa de presentación. Para configurar templates en `settings.py`, se debe definir el directorio donde se almacenarán los archivos HTML:



```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR / "templates"], # Directorio de 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',
            ],
        },
    },
]
```

Los archivos de templates deben guardarse en una carpeta llamada `templates/` dentro del proyecto.

Ejemplo de un template simple (`templates/index.html`):

```
<!DOCTYPE html>
<html lang="es">
<head>
    <title>Bienvenido a Django</title>
</head>
<body>
    <h1>¡Hola, mundo desde Django!</h1>
</body>
</html>
```

## Configuración de Paths en Django

Django permite manejar archivos estáticos y media a través de configuraciones en `settings.py`:

```
STATIC_URL = '/static/'
STATICFILES_DIRS = [BASE_DIR / "static"]

MEDIA_URL = '/media/'
MEDIA_ROOT = BASE_DIR / "media"
```

Con esta configuración, los archivos estáticos (CSS, JS, imágenes) deben almacenarse en la carpeta `static/`, mientras que los archivos subidos por los usuarios se guardarán en `media/`.

## Configuración de urls.py

Para definir las rutas de una aplicación, es necesario configurar el archivo `urls.py` tanto a nivel de proyecto como de aplicación.

### Configuración en urls.py del proyecto

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('mi_app/', include('mi_app.urls')), # Se enlaza con las URLs de la aplicación
]
```

### Configuración en urls.py de la aplicación



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

urlpatterns = [
    path('', views.index, name='index'),
]
```

#### Definición de una Vista en views.py

```
from django.http import HttpResponseRedirect

def index(request):
    return HttpResponseRedirect("¡Bienvenido a mi aplicación en Django!")
```

Cuando se accede a `http://127.0.0.1:8000/mi_app/`, se ejecutará la vista `index` y mostrará el mensaje en el navegador.

Última modificación: viernes, 2 de mayo de 2025, 19:55



## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](#). Desarrollado por [Moodle](#)



[Ver](#) [Hacer un envío](#)

## Ejercicio individual

### Objetivo

El estudiante llevará a cabo la instalación y configuración básica de Django en un entorno virtual, explorará su estructura y ejecutará su primer proyecto.

### Instrucciones

#### 1. Instalación de Django

- Investiga qué es `pip` y su función en la instalación de paquetes en Python.
- Crea un **entorno virtual** y activa la instalación de Django con `pip`.
- Comprobar que la instalación fue exitosa.

#### 2. Creación de un Proyecto Django

- Ejecuta el comando `django-admin startproject` para crear un nuevo proyecto.
- Examina la **estructura de carpetas** generada.

#### 3. Exploración del Proyecto

- Identifica los archivos clave dentro del proyecto (`settings.py`, `__init__.py`, `urls.py`).
- Investiga la función del archivo `manage.py` y su utilidad.

#### 4. Levantando el Servidor y Creación de un Superusuario

- Usa `python manage.py runserver` para iniciar el servidor de desarrollo.
- Explora la página web predeterminada de Django en el navegador.
- Crea un superusuario con `createsuperuser` y accede al panel de administración.

#### 5. Configuraciones Básicas

- Investiga cómo se configuran **templates**, **paths** y **urls** dentro del archivo `settings.py`.
- Realiza una configuración mínima de **templates** y **urls.py**.

#### 6. Documentación y Entrega

- Redacta un informe breve en el que describas los pasos seguidos, los comandos utilizados y lo aprendido en cada uno de ellos.

### Formato de Entrega

- **Archivo en PDF o documento de texto** con capturas de pantalla de los procesos clave.
- **Duración:** 1 jornada de clases.
- **Ejecución:** Individual

## Sumario de calificaciones

Ocultado a los estudiantes	No
Participantes	34
Enviados	0
Pendientes por calificar	0

?



## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](#). Desarrollado por [Moodle](#)

^

?

## Aprendizaje esperado 3

Implementar una aplicación web Django utilizando templates para el despliegue de páginas con contenido dinámico que dan solución a un requerimiento

### Sirviendo contenido estático: desplegar una página web estática

- [Introducción](#) al contenido estático en Django.
- Configuración y uso de archivos estáticos en el proyecto.

### Agregando un elemento navbar de Bootstrap

- Uso de Bootstrap para estructurar la navegación.
- Implementación de una barra de navegación reutilizable en la aplicación.

### Agregando un elemento jumbotron de Bootstrap

- Uso del componente Jumbotron para destacar información en la página.

### Agregando un elemento footer de Bootstrap

- Creación de un pie de página con Bootstrap y su integración en Django.

### Páginas responsivas

- Implementación de diseño adaptable con Bootstrap para mejorar la experiencia en distintos dispositivos.

### Links y navegación

- Configuración de enlaces internos en la aplicación con Django y Bootstrap.
- Uso de la etiqueta `url` en templates.

### El modelo MVC en Django

- Explicación del patrón Model-View-Controller en el contexto de Django.

### MVC vs MTV

- Diferencias entre el patrón MVC tradicional y el modelo MTV (Model-Template-View) de Django.

### Mostrar contenido estático con Bootstrap

- Integración de Bootstrap para mejorar la presentación de contenido estático en Django.

### Renderización dinámica y templates: ¿cómo usar páginas parciales?

- Uso de templates parciales para reutilizar componentes en distintas vistas.

### Vistas en Django

- Explicación del concepto de vistas en Django y su relación con los templates.

### Herencia de vistas

- Creación de vistas que comparten funcionalidad para mejorar la reutilización del código.

### Templates en Django

- [Introducción](#) al sistema de plantillas de Django y su estructura.

### Inclusión de templates

- Uso de `{% include %}` para dividir la interfaz en componentes reutilizables.



### Variables en plantillas: despliegue

- Uso de variables en templates para mostrar contenido dinámico.

### Iteradores: while, for

- Aplicación de bucles en plantillas para recorrer y mostrar listas de datos.

### Condiciones: if/elif/else

- Uso de estructuras condicionales en templates para personalizar la visualización de contenido.

### Aplicación de filtros

- Uso de filtros en templates para formatear y manipular datos.

### Contenido estático: STATIC\_URL, STATICFILES\_DIRS

- Configuración de archivos estáticos en Django para mejorar la organización del proyecto.

### Jerarquía de orden de llamadas de plantillas

- Explicación de cómo Django prioriza la carga de templates en una aplicación.

### Herencia de plantillas: bloques (block - endblock)

- Uso de bloques en templates para estructurar contenido de manera eficiente.

### Extends de plantillas

- Implementación de herencia de plantillas con { % extends % } para un diseño modular.

### Modificación de datos en plantillas hijas

- Personalización de contenido en plantillas heredadas sin modificar la estructura principal.

### Etiquetas URL en plantillas y redirecciónamiento

- Uso de { % url % } en templates para la navegación entre vistas.

### Generadores y from

- Explicación del uso de generadores en Django para optimizar el rendimiento.

### Manejo de errores

- [Introducción](#) a la gestión de errores en Django para mejorar la estabilidad del proyecto.

### Manejo de raise

- Uso de raise en Django para manejar excepciones en vistas.

### Creando modelos

- [Introducción](#) a la creación de modelos en Django y su integración con la base de datos.

Última modificación: viernes, 2 de mayo de 2025, 19:55



Contacta



✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](http://LMSACE.com). Desarrollado por [Moodle](http://Moodle)

^

?

## Contenido estático y Bootstrap

### Objetivos

- Servir contenido estático en Django.
- Integrar Bootstrap en una página estática.
- Agregar elementos como un **navbar**, un **jumbotron** y un **footer** utilizando Bootstrap.
- Crear páginas responsivas con Bootstrap.
- Manejar links y navegación dentro de una aplicación Django.
- Comprender el modelo **MVC** (Model-View-Controller) y su implementación en Django.
- Diferenciar entre **MVC** y **MTV** (Model-Template-View), el enfoque utilizado en Django.
- Mostrar contenido estático en Django utilizando **Bootstrap** para mejorar la presentación de una aplicación web.

### Configuración del contenido estático en Django

Django permite servir contenido estático como archivos CSS, JavaScript e imágenes a través de la configuración en `settings.py`.

```
STATIC_URL = '/static/'  
STATICFILES_DIRS = [BASE_DIR / "static"]
```

Se debe crear una carpeta `static/` en la raíz del proyecto para almacenar los archivos estáticos.

### Estructura recomendada

```
mi_proyecto/  
|__ mi_app/  
|__ static/  
|   |__ css/  
|   |__ js/  
|   |__ images/  
|__ templates/  
|__ manage.py
```

Para cargar archivos estáticos en un template, se usa `{% load static %}`.

```
{% load static %}  
<link rel="stylesheet" href="{% static 'css/estilos.css' %}">
```

### Desplegar una página web estática en Django

Para servir una página estática en Django, se crea una vista en `views.py`:

```
from django.shortcuts import render  
  
def pagina_estatica(request):  
    return render(request, 'index.html')
```

Luego, en `urls.py`, se define la ruta correspondiente:

?

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

urlpatterns = [
    path('', views.pagina_estatica, name='pagina_estatica'),
]
```

## Contenido estático: STATIC\_URL y STATICFILES\_DIRS

Django maneja archivos estáticos como imágenes, hojas de estilo CSS y scripts JavaScript a través de su sistema de archivos estáticos. Para configurarlos correctamente, es necesario definir dos parámetros clave en `settings.py`:

### STATIC\_URL

Este parámetro define la URL base desde donde Django servirá los archivos estáticos.

```
STATIC_URL = '/static/'
```

Con esta configuración, cualquier archivo ubicado en la carpeta `static/` del proyecto será accesible en la URL `/static/archivo.css`, por ejemplo.

### STATICFILES\_DIRS

Si se quiere almacenar archivos estáticos en un directorio específico dentro del proyecto, se debe agregar `STATICFILES_DIRS`:

```
import os
from pathlib import Path

BASE_DIR = Path(__file__).resolve().parent.parent

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

Con esta configuración, Django buscará archivos estáticos en la carpeta `static/` ubicada en la raíz del proyecto.

## Agregando un elemento Navbar de Bootstrap

Bootstrap proporciona componentes listos para usar. Un `navbar` se puede agregar al archivo `templates/index.html`:

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
    <div class="container">
        <a class="navbar-brand" href="#">Mi Sitio</a>
        <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNav">
            <span class="navbar-toggler-icon"></span>
        </button>
        <div class="collapse navbar-collapse" id="navbarNav">
            <ul class="navbar-nav">
                <li class="nav-item"><a class="nav-link" href="#">Inicio</a></li>
                <li class="nav-item"><a class="nav-link" href="#">Servicios</a></li>
                <li class="nav-item"><a class="nav-link" href="#">Contacto</a></li>
            </ul>
        </div>
    </div>
</nav>
```

Este `navbar` se adaptará automáticamente a dispositivos móviles gracias a Bootstrap.

## Agregando un elemento Jumbotron de Bootstrap



El **jumbotron** es un contenedor para resaltar información. Se puede agregar dentro del body en `index.html`:

```
<div class="container mt-4">
  <div class="p-5 mb-4 bg-light rounded-3">
    <div class="container-fluid py-5">
      <h1 class="display-5 fw-bold">Bienvenido a Mi Sitio</h1>
      <p class="fs-4">Este es un sitio web construido con Django y Bootstrap.</p>
      <a class="btn btn-primary btn-lg" href="#" role="button">Más información</a>
    </div>
  </div>
</div>
```

Este elemento ayuda a captar la atención del usuario con un diseño llamativo.

### Agregando un elemento Footer de Bootstrap

Un **footer** se coloca al final del `body` y se usa para mostrar información adicional.

```
<footer class="bg-dark text-white text-center py-3">
  <p>&copy; 2025 Mi Sitio. Todos los derechos reservados.</p>
</footer>
```

Este **footer** es fijo y se mantiene visible en la parte inferior de la página.

### Páginas responsivas con Bootstrap

Bootstrap facilita la creación de páginas adaptables a diferentes dispositivos. Se debe incluir el **meta viewport** en el `<head>` del HTML:

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

Ejemplo de diseño responsive usando el sistema de **grid** de Bootstrap:

```
<div class="container">
  <div class="row">
    <div class="col-md-6">
      <h2>Sección 1</h2>
      <p>Este contenido ocupa la mitad de la pantalla en dispositivos medianos.</p>
    </div>
    <div class="col-md-6">
      <h2>Sección 2</h2>
      <p>Otra sección que se adapta al tamaño de la pantalla.</p>
    </div>
  </div>
</div>
```

En pantallas pequeñas, las columnas se apilan verticalmente, mientras que en pantallas grandes se mantienen lado a lado.

### Links y navegación en Django

Para manejar navegación en Django, se usa la función `url` dentro de los templates.

Ejemplo de enlace dinámico en `index.html`:

```
<a href="{% url 'pagina_estatica' %}">Ir a la página principal</a>
```

Django permite generar menús dinámicos de navegación con clases activas:

```
<ul class="nav">
    <li class="nav-item"><a class="nav-link" {% if request.path == '/' %}active{% endif %}" href="{% url 'pagina-principal' %}>Inicio</a></li>
    <li class="nav-item"><a class="nav-link" href="#">Contacto</a></li>
</ul>
```

Esto ayuda a mejorar la experiencia de usuario resaltando la página activa.

### El modelo MVC en Django

El patrón **MVC** es un estándar en el desarrollo de aplicaciones web y se compone de tres partes:

- **Modelo (Model)**: Representa los datos y la lógica de negocio.
- **Vista (View)**: Muestra la información al usuario.
- **Controlador (Controller)**: Gestiona las interacciones del usuario y actualiza el modelo o la vista según sea necesario.

Django sigue este patrón pero con su propia variación llamada **MTV**.

### MVC vs MTV en Django

Aunque Django sigue la arquitectura **MVC**, su enfoque se denomina **MTV**:

MVC (Model-View-Controller)	MTV (Model-Template-View en Django)
<b>Modelo (Model)</b> : Maneja la lógica de datos	<b>Modelo (Model)</b> : Define las estructuras de datos y reglas de negocio.
<b>Vista (View)</b> : Muestra la información al usuario	<b>Template</b> : Genera el HTML para mostrar al usuario.
<b>Controlador (Controller)</b> : Gestiona la lógica de la aplicación	<b>Vista (View)</b> : Procesa las solicitudes, obtiene datos del modelo y los envía al template.

### Ejemplo en Django

Supongamos que queremos mostrar una lista de productos desde una base de datos.

#### 1. Modelo (models.py)

```
from django.db import models

class Producto(models.Model):
    nombre = models.CharField(max_length=100)
    precio = models.DecimalField(max_digits=10, decimal_places=2)
    descripcion = models.TextField()

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

#### 2. Vista (views.py)

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

def lista_productos(request):
    productos = Producto.objects.all()
    return render(request, 'productos.html', {'productos': productos})
```

#### 3. Template (templates/productos.html)

```
{% load static %}  
<!DOCTYPE html>  
<html lang="es">  
<head>  
    <link rel="stylesheet" href="{% static 'css/bootstrap.min.css' %}">  
    <title>Lista de Productos</title>  
</head>  
<body>  
    <div class="container mt-4">  
        <h1>Productos</h1>  
        <ul class="list-group">  
            {% for producto in productos %}  
                <li class="list-group-item">  
                    <strong>{{ producto.nombre }}</strong> - ${{ producto.precio }}  
                    <p>{{ producto.descripcion }}</p>  
                </li>  
            {% endfor %}  
        </ul>  
    </div>  
</body>  
</html>
```

#### 4. URL (urls.py)

```
from django.urls import path  
from . import views  
  
urlpatterns = [  
    path('productos/', views.lista_productos, name='lista_productos'),  
]
```

Última modificación: viernes, 2 de mayo de 2025, 19:55



## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](#). Desarrollado por [Moodle](#)

?

## Renderización dinámica y plantillas en Django

### Objetivos

- Comprender el funcionamiento de los templates en Django.
- Implementar herencia de vistas para reutilizar código.
- Utilizar variables, iteradores y condiciones en plantillas.
- Aplicar filtros en los templates de Django.

### ¿Cómo usar páginas parciales?

Django permite estructurar las páginas web en componentes reutilizables, como un navbar, un footer o una barra lateral. Estos elementos pueden definirse en archivos separados y luego incluirse en otras plantillas con `{% include %}`.

Ejemplo de uso de un archivo `navbar.html`:

```
<nav>
    <ul>
        <li><a href="{% url 'home' %}">Inicio</a></li>
        <li><a href="{% url 'about' %}">Sobre nosotros</a></li>
    </ul>
</nav>
```

Este `navbar.html` se puede incluir en otras plantillas con:

```
{% include 'navbar.html' %}
```

Esto evita la duplicación de código y facilita el mantenimiento.

### Vistas en Django

Las vistas en Django manejan la lógica detrás de cada página web. Se definen en `views.py` y pueden devolver respuestas HTML, JSON u otros formatos.

Ejemplo de una vista en `views.py`:

```
from django.shortcuts import render

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

Esta vista renderiza el template `home.html` cuando se accede a la URL correspondiente.

### Herencia de vistas

Django permite reutilizar código HTML mediante la herencia de plantillas. Se usa un archivo base con la estructura principal y se extiende en otras páginas.

Ejemplo de `base.html`:



```
<!DOCTYPE html>
<html lang="es">
<head>
    <title>{% block title %}Mi sitio{% endblock %}</title>
</head>
<body>
    {% include 'navbar.html' %}
    <main>
        {% block content %}{% endblock %}
    </main>
</body>
</html>
```

En otra plantilla, se puede extender `base.html` y definir el contenido específico:

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

{% block title %}Inicio{% endblock %}

{% block content %}
    <h1>Bienvenido a mi sitio</h1>
{% endblock %}
```

Esto permite modificar solo partes específicas sin duplicar código.

## Templates en Django

Los templates son archivos HTML con una sintaxis especial de Django. Permiten insertar variables, estructuras de control y filtros.

Ejemplo de template con variables:

```
<h1>Hola, {{ nombre }}</h1>
```

Si en la vista se pasa un contexto con `{ "nombre": "Carlos" }`, el resultado será:

```
<h1>Hola, Carlos</h1>
```

Las variables se pasan desde la vista y se utilizan dentro de los templates con `{{ variable }}`.

Ejemplo en `views.py`:

```
def perfil(request):
    usuario = {"nombre": "Carlos", "edad": 25}
    return render(request, 'perfil.html', usuario)
```

## Inclusión de templates

Además del `{% include %}` para páginas parciales, Django permite incluir otras plantillas dinámicamente según las necesidades del usuario.

Ejemplo:

```
{% if usuario.is_authenticated %}
    {% include 'perfil.html' %}
{% else %}
    {% include 'login.html' %}
{% endif %}
```

## Iteradores en plantillas

Django permite recorrer listas en los templates con `{% for %}`.

Ejemplo:

```
<ul>
    {% for producto in productos %}
        <li>{{ producto.nombre }} - {{ producto.precio }}</li>
    {% endfor %}
</ul>
```

Si `productos` es una lista de objetos en la vista, se mostrarán dinámicamente en el HTML.

Para iteraciones con límite, se usa `forloop.counter`:

```
<p>Producto {{ forloop.counter }}: {{ producto.nombre }}</p>
```

También se puede usar `{% while %}` con condiciones simples.

## Condiciones en templates

Las estructuras de control condicional permiten modificar la visualización.

Ejemplo con `{% if %}`:

```
{% if usuario.is_authenticated %}
    <p>Bienvenido, {{ usuario.nombre }}</p>
{% else %}
    <p>Por favor, inicia sesión</p>
{% endif %}
```

Se pueden agregar más condiciones con `{% elif %}` y `{% else %}`.

Ejemplo con varias condiciones:

```
{% if edad < 18 %}
    <p>Eres menor de edad</p>
{% elif edad >= 18 and edad < 65 %}
    <p>Eres adulto</p>
{% else %}
    <p>Eres adulto mayor</p>
{% endif %}
```

## Aplicación de filtros en Django

Django proporciona filtros para transformar variables dentro de los templates. Algunos ejemplos:

- `{{ nombre|lower }}` → Convierte a minúsculas.
- `{{ nombre|upper }}` → Convierte a mayúsculas.
- `{{ lista|length }}` → Muestra la cantidad de elementos.

Ejemplo en una plantilla:

```
<p>Tu nombre en mayúsculas: {{ nombre|upper }}</p>
<p>Cantidad de productos: {{ productos|length }}</p>
```

Se pueden combinar varios filtros:



```
<p>Nombre capitalizado: {{ nombre|lower|capfirst }}</p>
```

Última modificación: viernes, 2 de mayo de 2025, 19:56



## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](#). Desarrollado por [Moodle](#)



## Más sobre plantillas en Django

### Objetivos

- Comprender cómo se maneja la jerarquía de las plantillas en Django.
- Aprender a usar la herencia de plantillas para mejorar la reutilización del código.
- Entender cómo modificar datos en plantillas hijas.
- Implementar redirecciones y etiquetas URL en las plantillas de Django.
- Conocer el manejo de errores y excepciones con `raise`.

### Etiquetas URL en plantillas y redireccionamiento

En Django, las URLs se gestionan mediante el sistema de `url` en el archivo `urls.py`. Para generar enlaces de manera dinámica en las plantillas, se utiliza la etiqueta `{% url %}`, que crea un enlace a una vista en función de su nombre.

Ejemplo:

```
<a href="{% url 'home' %}">Ir a la página principal</a>
```

Este código genera un enlace a la vista `home` definida en `urls.py`.

También es posible redirigir desde una vista a otra con el método `redirect()`:

```
from django.shortcuts import redirect

def redirigir_a_home(request):
    return redirect('home')
```

### Generadores y from

Django permite importar funciones y módulos en plantillas utilizando la etiqueta `{% from %}`. Esto es útil para importar filtros o funciones que se usen en la plantilla.

Ejemplo de uso de `from`:

```
{% from 'my_tags' import mi_funcion %}
```

Esto permite utilizar `mi_funcion` dentro de la plantilla.

### Manejo de errores

Django proporciona herramientas para gestionar errores de manera controlada, a través de excepciones. El manejo de errores se realiza con la instrucción `try` y `except`.

Ejemplo de manejo de errores en una vista:

```
def vista(request):
    try:
        resultado = 10 / 0 # Esto generará una excepción
    except ZeroDivisionError:
        resultado = "No se puede dividir por cero"
    return render(request, 'template.html', {'resultado': resultado})
```

### Manejo de raise

La instrucción `raise` en Python se utiliza para generar excepciones manualmente, lo que puede ser útil para asegurar que se cumplan ciertas condiciones.

Ejemplo:

```
def verificar_edad(edad):
    if edad < 18:
        raise ValueError("Edad no válida, debe ser mayor de 18")
    return "Edad válida"
```

Cuando la edad es menor de 18, se lanza un `ValueError` que interrumpe la ejecución del programa.

### Jerarquía de orden de llamadas de plantillas

Django sigue un orden de llamadas para procesar las plantillas, que permite organizar y reutilizar plantillas de manera eficiente. Cuando se hace una petición, Django busca primero las plantillas en las rutas especificadas dentro de los directorios de las apps y en las configuraciones de `DIRS` dentro de `settings.py`. Si no encuentra la plantilla en esos lugares, buscará en las plantillas de las apps instaladas.

Ejemplo de búsqueda de plantillas en `settings.py`:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR / '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',
            ],
        },
    },
]
```

### Herencia de plantillas: Bloques (Block - Endblock)

La herencia de plantillas en Django permite crear una estructura base que puede ser extendida por otras plantillas, reduciendo la duplicación de código. Para esto se utilizan los bloques `{% block %}` y `{% endblock %}`.

Ejemplo de plantilla base `base.html`:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <title>{% block title %}Mi sitio web{% endblock %}</title>
</head>
<body>
    <header>
        <h1>{% block header %}Bienvenido{% endblock %}</h1>
    </header>
    <main>
        {% block content %}{% endblock %}
    </main>
    <footer>
        {% block footer %}Derechos reservados{% endblock %}
    </footer>
</body>
</html>
```

En una plantilla hija, se puede extender la plantilla base y sobrescribir los bloques definidos:

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

{% block title %}Página de inicio{% endblock %}

{% block content %}
    <h2>Contenido principal</h2>
    <p>Bienvenido a la página de inicio.</p>
{% endblock %}
```

### Extends de plantillas

El uso de `{% extends %}` permite incluir una plantilla base en otra, de manera que toda la estructura del HTML se hereda. Esto facilita la organización y evita duplicaciones de código.

Ejemplo:

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

{% block content %}
    <h2>Bienvenido al panel de usuario</h2>
{% endblock %}
```

### Modificación de datos en plantillas hijas

En Django, es posible modificar y personalizar datos dentro de las plantillas hijas. Esto se logra con los bloques definidos en la plantilla base, que pueden sobrescribirse en las plantillas hijas.

Ejemplo:

En la plantilla base, se define un bloque de contenido:

```
{% block content %}
    <p>Contenido por defecto</p>
{% endblock %}
```

Y en la plantilla hija, se sobrescribe este bloque:

```
{% block content %}
    <p>Contenido personalizado para la página específica.</p>
{% endblock %}
```

Última modificación: viernes, 2 de mayo de 2025, 19:56





## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](#). Desarrollado por [Moodle](#)

^

?

## Modelos en Django

### Objetivos

- Entender qué son los modelos en Django y su función dentro de la estructura de una aplicación.
- Aprender a definir modelos sin conexión a la base de datos.
- Conocer cómo Django maneja la creación de modelos y la representación de datos sin necesidad de conectarse a una base de datos en este momento.

### Qué son los modelos en Django

En Django, los **modelos** son representaciones de las tablas de una base de datos. Cada modelo es una clase que hereda de `django.db.models.Model` y define la estructura de los datos que queremos almacenar en una tabla. Aunque la principal finalidad de los modelos es interactuar con la base de datos, también se utilizan para estructurar la lógica de negocio de la aplicación.

#### Ejemplo básico de un modelo:

```
from django.db import models

class Persona(models.Model):
    nombre = models.CharField(max_length=100)
    edad = models.IntegerField()
    email = models.EmailField()

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

En este ejemplo, el modelo `Persona` tiene tres campos: `nombre`, `edad` y `email`. Django crearía automáticamente una tabla en la base de datos para almacenar estos datos, con columnas correspondientes a cada uno de los atributos.

### Definición de modelos

Aunque el propósito de los modelos es interactuar con una base de datos, puedes definir modelos en Django incluso sin haber configurado una base de datos. Esta es una práctica útil cuando aún no estás listo para conectarte a una base de datos pero deseas definir la estructura de los datos de tu aplicación.

La principal diferencia es que, al no estar conectados a la base de datos, no podrás ejecutar migraciones para crear las tablas ni guardar datos de manera persistente, pero aún puedes definir las estructuras de los modelos y trabajar con ellos a nivel de código.

#### Ejemplo de definición de un modelo:

```
from django.db import models

class Producto(models.Model):
    nombre = models.CharField(max_length=100)
    precio = models.DecimalField(max_digits=10, decimal_places=2)
    disponible = models.BooleanField(default=True)

    def __str__(self):
        return f"{self.nombre} - ${self.precio}"
```

En este ejemplo, `Producto` es un modelo que define un producto con nombre, precio y disponibilidad. Al no estar conectado a una base de datos, puedes realizar pruebas, validaciones o documentación sin afectar ninguna base de datos real.

## ¿Qué sucede cuando no hay conexión a la base de datos?

Cuando no conectas tu proyecto Django a una base de datos, los modelos que definas no tendrán ninguna representación persistente. Esto significa que:

1. **No se crearán tablas** en una base de datos para almacenar los datos.
2. **No se podrán realizar migraciones**, que son necesarias para reflejar los cambios en la base de datos.
3. **No se podrá guardar ni recuperar información** de una base de datos.

Sin embargo, el uso de modelos sigue siendo útil porque puedes definir la estructura de los datos y probar el comportamiento de las clases de modelo sin tener que configurar una base de datos desde el principio.

## Crear una vista para pasar los datos del modelo a la plantilla

Las vistas en Django son las encargadas de recibir las solicitudes del usuario, consultar los datos de la base de datos (o de los modelos) y devolver una respuesta, generalmente en forma de una plantilla HTML.

### Vista basada en función (function-based view)

En Django, puedes utilizar **vistas basadas en funciones** para pasar los datos a la plantilla. Para hacer esto, puedes usar la función `render()` para renderizar la plantilla HTML y pasarte los datos.

```
# views.py
from django.shortcuts import render

def lista_productos(request):
    # Crear datos de ejemplo de productos
    productos = [
        {"nombre": "Producto A", "precio": 100.50, "disponible": True},
        {"nombre": "Producto B", "precio": 200.75, "disponible": False},
        {"nombre": "Producto C", "precio": 50.99, "disponible": True},
    ]

    # Pasar los productos a la plantilla
    return render(request, 'productos/lista.html', {'productos': productos})
```

## Crear la plantilla HTML para renderizar los datos del modelo

La plantilla seguirá siendo la misma que te mostré anteriormente. Aquí no cambiamos nada, ya que seguimos mostrando los datos de la lista que hemos pasado desde la vista.

**Plantilla lista.html:**



```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Lista de Productos</title>
</head>
<body>
    <h1>Productos Disponibles</h1>
    <ul>
        {% for producto in productos %}
            <li>
                <strong>{{ producto.nombre }}</strong> - ${{ producto.precio }}
                {% if producto.disponible %}
                    <span>(Disponible)</span>
                {% else %}
                    <span>(No disponible)</span>
                {% endif %}
            </li>
        {% empty %}
            <p>No hay productos disponibles.</p>
        {% endfor %}
    </ul>
</body>
</html>
```

## Configurar la URL

Asegúrate de que la URL esté configurada para acceder a esta vista. En `urls.py`:

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

urlpatterns = [
    path('productos/', views.lista_productos, name='lista_productos'),
]
```

## Acceder a la Vista desde el Navegador

Una vez que hayas configurado la vista, la plantilla y la URL, puedes acceder a la lista de productos de ejemplo desde el navegador ingresando la URL `/productos/`.

Última modificación: viernes, 2 de mayo de 2025, 19:56



Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)



Copyright © 2017 -Desarrollado por [LMSACE.com](http://LMSACE.com). Desarrollado por [Moodle](http://Moodle)

^

?

## Ejercicio individual

### Objetivo

El objetivo de esta actividad es que los estudiantes creen una página web estática utilizando Django y Bootstrap, incorporando diversos componentes como un **navbar**, un **jumbotron**, un **footer**, y trabajando con vistas dinámicas, templates, y contenido estático.

### Instrucciones Generales

1. **Diseño de la Página:** Crea una página principal para tu sitio web que incluya los siguientes elementos:
  - **Navbar:** Agrega un elemento de navegación (navbar) utilizando Bootstrap.
  - **Jumbotron:** Incluye un jumbotron en la parte superior de la página para destacar algún contenido importante.
  - **Footer:** Agrega un pie de página (footer) con información adicional, como contacto o derechos reservados.
2. **Responsividad:** Asegúrate de que la página sea **responsiva**, utilizando las clases adecuadas de Bootstrap para que se adapte correctamente a diferentes tamaños de pantalla.
3. **Navegación:** La página debe contener enlaces (links) de navegación en el **navbar** que redirijan a diferentes secciones o páginas dentro de tu sitio web (aunque estas pueden ser simples vistas estáticas para esta actividad).
4. **Modelo MVC en Django:** Implementa el patrón **MVC** (Modelo-Vista-Controlador) en tu aplicación Django, pero recuerda que Django sigue el patrón **MTV** (Modelo-Template-Vista). Asegúrate de:
  - **Modelo:** Definir algún modelo simple (si es necesario) para tu aplicación.
  - **Vista:** Crear vistas que rendericen las páginas estáticas con el contenido necesario.
  - **Template:** Utilizar templates de Django para estructurar el HTML de las vistas.
5. **Renderización Dinámica:** Utiliza **templates** de Django para renderizar contenido dinámico dentro de la página. Asegúrate de incluir algunas **variables** en los templates que sean pasadas desde las vistas (por ejemplo, el título de la página, algún mensaje dinámico, etc.).
6. **Herencia de Templates:** Utiliza **herencia de templates** para organizar tu código y evitar la repetición de estructuras comunes (como el **navbar** y el **footer**) en todas las páginas.
7. **Condiciones e Iteradores:** En alguna parte de tu sitio web, implementa **condiciones** (if/else) y **iteradores** (for) para mostrar contenido de manera dinámica. Por ejemplo, muestra una lista de elementos que podría ser dinámica dependiendo de los datos pasados desde la vista.
8. **Manejo de Contenido Estático:**
  - Asegúrate de que tu **CSS** y **archivos estáticos** (como imágenes o JavaScript) sean cargados correctamente.
  - Utiliza **STATIC\_URL** y **STATICFILES\_DIRS** para configurar y cargar archivos estáticos.
9. **Manejo de Errores:** Implementa un manejo de errores simple usando el bloque **try/except** de Python para manejar cualquier error común que pueda surgir durante el proceso de desarrollo.

### Entrega

- Un archivo zip con los archivos de tu proyecto o un repositorio de Github.
- Ejecución: Individual.
- Duración: 1 jornada de clases.

## Sumario de calificaciones



Ocultado a los estudiantes	No	?
Participantes	34	?

Enviados	0
Pendientes por calificar	0



## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](#). Desarrollado por [Moodle](#)



[Ver](#) [Hacer un envío](#)

## Ejercicio grupal

### Objetivo

El objetivo de este ejercicio grupal es que los estudiantes trabajen en equipo para construir un sitio web dinámico de recetas utilizando Django. Cada miembro del equipo será responsable de una parte del proyecto, aplicando los conceptos de MVC/MTV, navegación, contenido estático y dinámico, y diseño responsive utilizando Bootstrap.

### Contexto del Proyecto

**La App de Recetas** es un sitio web donde los usuarios pueden ver y explorar diversas recetas de cocina. El proyecto debe tener una página de inicio con una lista de recetas, una página para cada receta individual con sus detalles, y una página de contacto. El sitio debe ser visualmente atractivo, utilizar componentes de Bootstrap y ser completamente funcional.

### Requisitos del Proyecto

#### 1. Estructura del Proyecto:

- **Navbar:** Una barra de navegación que permita acceder a las diferentes secciones del sitio web: "Inicio", "Recetas", "Contacto".
- **Jumbotron:** Un jumbotron en la página de inicio que resalte el propósito del sitio y de qué trata.
- **Footer:** Un pie de página con información adicional sobre el sitio web y los derechos de autor.

#### 2. Páginas Estáticas y Dinámicas:

- **Página de Inicio:** Debe mostrar un mensaje de bienvenida y una lista de las últimas recetas (cada receta debe mostrar el nombre y una breve descripción).
- **Página de Recetas:** Cada receta debe tener una página individual que muestre los detalles de la receta (nombre, ingredientes, instrucciones de preparación). Estas páginas deben ser dinámicas, mostrando la información desde la base de datos.
- **Página de Contacto:** Un formulario de contacto que permita a los usuarios enviar un mensaje.

#### 3. Navegación:

- Cada página debe estar vinculada a través de una barra de navegación (Navbar). Asegúrate de que los enlaces de la navegación funcionen correctamente y dirijan al usuario a las páginas correspondientes.
- El sitio debe ser **responsive**, por lo que debe ajustarse bien a dispositivos móviles y de escritorio.

#### 4. Contenido Estático:

- **Imágenes de las recetas:** Cada receta debe incluir una imagen estática. Los archivos de imagen deben estar almacenados en la carpeta de archivos estáticos de Django.
- **Archivos CSS personalizados:** Además de los estilos de Bootstrap, puedes crear un archivo CSS personalizado para darle un toque único al sitio.

#### 5. Modelos y Vistas:

- Crea un modelo de **Receta** con los siguientes campos:
  - nombre (CharField)
  - ingredientes (TextField)
  - instrucciones (TextField)
  - imagen (ImageField)
- Utiliza **vistas genéricas** para mostrar las recetas. La página de inicio debe mostrar una lista de las recetas y, al hacer clic en una receta, el usuario debe ser redirigido a la página de detalles de la receta.

#### 6. Plantillas (Templates):

- Utiliza **herencia de templates** para evitar la repetición de código. Por ejemplo, el navbar y el footer deben

?

estar incluidos en un template base, y las demás páginas deben extender este template base.

- En la página de **Recetas**, utiliza **iteradores** (como un ciclo **for**) para recorrer las recetas y mostrarlas en la página.
- En la página de **Detalles de Receta**, usa condiciones (**if/else**) para mostrar la receta solo si existe en la base de datos.

#### 7. Redirección y Manejo de Errores:

- Implementa un manejo de errores adecuado. Si una receta no existe, el usuario debe ser redirigido a una página de error personalizada.
- Si el usuario intenta acceder a una página de contacto sin completar el formulario, debe recibir un mensaje de advertencia.

#### 8. Enlaces de URL y Redirección:

- Utiliza **etiquetas URL** en los templates para hacer las URLs dinámicas.
- Al enviar un formulario de contacto, redirige al usuario a una página de confirmación de mensaje enviado.

#### Entrega

- Entregar un archivo zip con los archivos del proyecto o un Github.
- Duración: 1 jornada de clases.
- Ejecución: Grupal.

## Sumario de calificaciones

Ocultado a los estudiantes	No
Participantes	34
Enviados	0
Pendientes por calificar	0



## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](http://LMSACE.com). Desarrollado por [Moodle](http://Moodle)



## Aprendizaje esperado 4

**Implementar formularios en un aplicativo web utilizando framework Django para la captura y procesamiento de información dando solución a un problema**

### Formularios en Django

Los formularios en Django permiten la captación de datos a través de una interfaz web. Django proporciona herramientas potentes y flexibles para trabajar con formularios y manejar el procesamiento de datos.

#### Forms de Django vs Formularios HTML

- Django proporciona un enfoque más estructurado, manejando validaciones, errores, y otros aspectos automáticamente.
- Los formularios HTML requieren un mayor trabajo manual para la validación y manejo de errores. Django simplifica estos procesos.

#### Enlace con la vista

Un formulario se conecta a una vista en Django. El formulario puede ser utilizado en una vista para renderizar datos, validar y capturar la información del usuario.

#### La FormClass de Django

Django ofrece la clase `Form` para facilitar la creación y validación de formularios. Esta clase puede ser extendida para personalizar el formulario de acuerdo con las necesidades del proyecto.

#### Procesamiento de formularios

Una vez que el formulario es enviado, se procesan los datos en la vista. Django maneja la validación y el procesamiento de la entrada del formulario, proporcionando una forma fácil de recuperar y usar esos datos.

#### Construyendo un formulario

Para crear un formulario en Django, se define una clase que hereda de `django.forms.Form` y se asignan los campos que se desean capturar.

#### Agregando la vista del formulario

La vista de un formulario maneja las solicitudes GET y POST. El formulario se renderiza inicialmente a través de GET, y luego los datos son procesados cuando el formulario es enviado (POST).

#### Agregando el template de despliegue

Una vez que la vista del formulario está lista, se agrega el HTML correspondiente para mostrar el formulario al usuario en la interfaz. Este template se encarga de mostrar los campos y los posibles errores.

#### Plantillas de formularios

Django facilita la creación de plantillas que se pueden reutilizar, lo que ahorra tiempo y esfuerzo al crear formularios en distintas partes de la aplicación.

#### Renderizar los campos de forma manual

Aunque Django proporciona una forma automática de renderizar formularios, también es posible renderizar cada campo de manera manual en la plantilla para personalizar su apariencia.

#### Plantillas de formulario reutilizables

Se pueden crear formularios que sean reutilizables en diferentes partes del proyecto. Esta reutilización ayuda a mantener el código limpio y escalable.

## Acerca de InlineFormsets y vinculación por llaves foráneas

Los **InlineFormsets** son útiles cuando se trabaja con relaciones de bases de datos que tienen claves foráneas. Permiten manejar formularios relacionados en un único formulario.

## Mensaje de errores, Models y Binding

Los formularios en Django pueden mostrar mensajes de error si los datos no son válidos. Además, Django permite el enlace entre formularios y modelos para capturar información directamente en la base de datos.

Última modificación: viernes, 2 de mayo de 2025, 19:57



## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](http://LMSACE.com). Desarrollado por [Moodle](http://Moodle)



## Formulario en Django

### Objetivos

- Entender la diferencia entre los formularios en Django y los formularios HTML tradicionales.
- Aprender a enlazar formularios con las vistas y plantillas en Django.
- Conocer el proceso de construcción y procesamiento de formularios en Django.
- Crear formularios reutilizables y manejar errores y validaciones en formularios.

### 1. Formularios en Django

Django ofrece una manera sencilla de manejar formularios a través del uso de clases `Form` que proporcionan una capa de abstracción sobre los formularios HTML tradicionales.

#### 1.1. Forms de Django vs Formularios HTML

Los formularios HTML requieren la creación manual de etiquetas `<form>`, `<input>`, `<textarea>`, y su respectivo manejo de datos en el servidor. Django, por otro lado, ofrece una clase `Form` que abstrae todo este proceso y permite la validación y manejo de datos de manera sencilla.

- **Formularios HTML:** Necesitas escribir todo el código HTML de cada campo y gestionarlo manualmente en la vista.
- **Forms de Django:** Usas una clase `Form` en Python para definir los campos y validaciones, y Django se encarga del resto (como generar los campos HTML correspondientes).

### 2. Enlace con la Vista

Para trabajar con formularios en Django, debes vincular la clase `Form` con una vista que gestione la lógica de presentación y procesamiento del formulario.

#### 2.1. La FormClass de Django

Django tiene una clase base `forms.Form`, que se utiliza para crear formularios. La clase define los campos que se van a incluir y puede incluir validaciones.

##### Ejemplo de clase Form:

```
# forms.py
from django import forms

class ContactoForm(forms.Form):
    nombre = forms.CharField(max_length=100)
    correo = forms.EmailField()
    mensaje = forms.CharField(widget=forms.Textarea)
```

#### 2.2. Procesamiento de Formularios

Django ofrece un manejo sencillo del procesamiento de formularios. Esto implica comprobar si el formulario ha sido enviado, validado y procesado correctamente.

##### Ejemplo de procesamiento en la vista:



```
# views.py
from django.shortcuts import render
from .forms import ContactoForm

def contacto(request):
    if request.method == 'POST':
        form = ContactoForm(request.POST)
        if form.is_valid():
            # Procesar datos
            nombre = form.cleaned_data['nombre']
            correo = form.cleaned_data['correo']
            mensaje = form.cleaned_data['mensaje']
            # Realizar alguna acción con los datos, como enviar un correo
            return render(request, 'contacto_exito.html', {'nombre': nombre})
    else:
        form = ContactoForm()

    return render(request, 'contacto.html', {'form': form})
```

### 3. Construyendo un Formulario

Los formularios de Django se pueden construir utilizando los campos proporcionados en la clase `Form`. Cada campo se representa como una clase de campo, como `CharField`, `EmailField`, `IntegerField`, entre otros.

#### 3.1. Agregando la Vista del Formulario

El formulario debe ser gestionado en una vista que lo procese y lo pase al template para su visualización.

##### Ejemplo de vista que renderiza el formulario:

```
# views.py
def formulario(request):
    if request.method == 'POST':
        form = ContactoForm(request.POST)
        if form.is_valid():
            # Procesar datos
            return render(request, 'formulario_exito.html')
    else:
        form = ContactoForm()

    return render(request, 'formulario.html', {'form': form})
```

#### 3.2. Agregando el Template de Despliegue

El template es el archivo HTML donde se presenta el formulario. Django se encarga de renderizar los campos y mostrar los errores.

##### Ejemplo de template `formulario.html`:

```
<form method="POST">
    {% csrf_token %}
    {{ form.as_p }} <!-- Muestra el formulario con etiquetas <p> por cada campo -->
    <button type="submit">Enviar</button>
</form>
```

### 4. Plantillas de Formularios

En lugar de escribir los formularios manualmente, Django permite usar las plantillas de formulario para renderizarlos fácilmente.

#### 4.1. Renderizar los Campos de Forma Manual

?

Aunque {{ form.as\_p }} es una forma rápida de mostrar el formulario, a veces necesitas un control más fino sobre cómo se presentan los campos. Esto se puede hacer renderizando los campos de forma manual.

#### Ejemplo de renderizado manual:

```
<form method="POST">
    {% csrf_token %}
    <label for="{{ form.nombre.id_for_label }}">Nombre:</label>
    {{ form.nombre }}<br>

    <label for="{{ form.correo.id_for_label }}">Correo:</label>
    {{ form.correo }}<br>

    <label for="{{ form.mensaje.id_for_label }}">Mensaje:</label>
    {{ form.mensaje }}<br>

    <button type="submit">Enviar</button>
</form>
```

#### 4.2. Plantillas de Formulario Reutilizables

Para reutilizar formularios en diferentes partes del proyecto, puedes crear plantillas de formularios reutilizables que incluyan la misma estructura básica.

#### Ejemplo de plantilla reusable (formulario\_base.html):

```
<form method="POST">
    {% csrf_token %}
    {% block formulario_campo %}
    {{ form.as_p }}
    {% endblock %}
    <button type="submit">Enviar</button>
</form>
```

Luego puedes extenderla en otras plantillas:

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

{% block formulario_campo %}
    <label for="{{ form.nombre.id_for_label }}">Nombre:</label>
    {{ form.nombre }}<br>
    {{ form.as_p }}
    {% endblock %}
```

#### 5. Acerca de InlineFormSets y Vinculación por Llaves Foráneas

Un `InlineFormSet` es un conjunto de formularios que permiten manejar modelos relacionados con claves foráneas (Foreign Keys).

##### 5.1. Usar `InlineFormSets`

`InlineFormSets` son útiles cuando tienes modelos relacionados y necesitas crear formularios que involucren ambos modelos.

#### Ejemplo de `InlineFormSet`:



```

from django.forms import modelformset_factory
from .models import Producto, Categoria

def producto_categoria(request):
    ProductoFormSet = modelformset_factory(Producto, fields=('nombre', 'precio', 'categoria'))
    if request.method == 'POST':
        formset = ProductoFormSet(request.POST)
        if formset.is_valid():
            formset.save()
    else:
        formset = ProductoFormSet(queryset=Producto.objects.none())

    return render(request, 'producto_categoria.html', {'formset': formset})

```

## 6. Mensaje de Errores, Models y Binding

Django maneja automáticamente los errores en los formularios y los muestra junto a los campos correspondientes.

### 6.1. Mostrar Errores de Validación

Si un formulario no es válido, puedes mostrar los errores de forma sencilla.

#### Ejemplo de mostrar errores:

```

<form method="POST">
    {% csrf_token %}
    {{ form.as_p }}
    {% if form.errors %}
        <div class="errors">
            <ul>
                {% for field in form %}
                    {% for error in field.errors %}
                        <li>{{ error }}</li>
                    {% endfor %}
                {% endfor %}
            </ul>
        </div>
    {% endif %}
    <button type="submit">Enviar</button>
</form>

```

### 6.2 Binding en Formularios

Django hace automáticamente el "binding" (vinculación) de los datos del formulario con los campos del modelo, lo que facilita la persistencia de datos en la base de datos cuando se utiliza con `ModelForm`.

#### Ejemplo de ModelForm:

```

# forms.py
from django import forms
from .models import Producto

class ProductoForm(forms.ModelForm):
    class Meta:
        model = Producto
        fields = ['nombre', 'precio', 'categoria']

```

Última modificación: viernes, 2 de mayo de 2025, 19:57





## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](#). Desarrollado por [Moodle](#)

^

?

---

---

^

?

## Ejercicio grupal

### Objetivo

El objetivo de esta actividad es que los estudiantes trabajen en equipo para desarrollar una aplicación en Django que permita a los usuarios registrar eventos. A través de este ejercicio, los estudiantes practicarán la creación, validación y procesamiento de formularios en Django, utilizando plantillas reutilizables y vinculándolo con Modelos.

### Contexto

**Aplicación de Registro de Eventos:** Se desarrollará una aplicación en la que los usuarios puedan registrar eventos con información como el nombre del evento, fecha, ubicación y una lista de participantes. Cada participante podrá ingresar su nombre y correo electrónico.

### Requisitos

#### 1. Formulario de Registro de Eventos:

- El formulario debe permitir registrar los siguientes datos:
  - **Nombre del Evento** (CharField, obligatorio)
  - **Fecha** (DateField, obligatorio)
  - **Ubicación** (CharField, opcional)
- Debe incluir validaciones:
  - La fecha del evento debe ser futura.
  - El nombre del evento no debe superar los 100 caracteres.

#### 2. Formulario para Participantes:

- Se debe incluir un formulario para agregar múltiples participantes al evento, con los siguientes campos:
  - **Nombre del Participante** (CharField, obligatorio)
  - **Correo Electrónico** (EmailField, obligatorio)

#### 3. Uso de FormClass de Django:

- Crear una FormClass para el formulario de eventos.
- Crear una FormClass para el formulario de participantes.
- Enlazar ambos formularios con la vista correspondiente.

#### 4. Vistas y Templates:

- Crear una vista para manejar la solicitud GET (mostrar los formularios) y la solicitud POST (procesar los datos).
- Mostrar errores de validación debajo de los campos incorrectos.
- Utilizar plantillas reutilizables para el formulario de eventos y el de participantes.

#### 5. Validación y Manejo de Errores:

- Si el formulario no es válido, mostrar mensajes de error adecuados.
- Si el registro es exitoso, mostrar un mensaje de confirmación.

#### 6. Uso de Plantillas Reutilizables:

- Diseñar un template base para los formularios.
- Usar bloques en la plantilla para modificar el contenido de cada formulario específico.

### Entrega

- Entregar un archivo zip con todos los archivos del proyecto o un repositorio Github.
- Duración: 1 jornada de clases
- Ejecución: Grupal.



## Sumario de calificaciones

Ocultado a los estudiantes	No	?
----------------------------	----	---

<b>Participantes</b>	34
<b>Enviados</b>	0
<b>Pendientes por calificar</b>	0



## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](#). Desarrollado por [Moodle](#)



## Aprendizaje esperado 5

**Implementar mecanismos de autenticación y autorización para el establecimiento de controles de seguridad utilizando el framework Django.**

### ¿Qué es el modelo Auth de Django?

El modelo Auth de Django es un sistema integrado para manejar la autenticación y la autorización de usuarios. Incluye funcionalidades como el registro de usuarios, inicio de sesión, y gestión de contraseñas de forma segura.

### ¿Cómo maneja la seguridad Django?

Django proporciona un sistema de seguridad robusto que incluye protección contra CSRF (Cross-Site Request Forgery), XSS (Cross-Site Scripting), y protección contra ataques de inyección SQL, entre otros. Además, ofrece una gestión automática de contraseñas y autenticación de usuario con encriptación segura.

### ¿Cómo utilizamos el modelo Auth de Django?

El modelo Auth de Django se usa principalmente para gestionar la autenticación de los usuarios. Puedes utilizar sus funciones para crear usuarios, asignarles contraseñas seguras, verificar credenciales y administrar sesiones de usuario. El modelo incluye varias clases como `User` y `Group` para trabajar con roles y permisos.

### Ejecutando las migraciones

Para que los cambios en el modelo Auth se reflejen en la base de datos, es necesario ejecutar migraciones. Django se encarga de gestionar las tablas relacionadas con la autenticación y otras configuraciones de seguridad.

### Enrutamiento Login/Logout

Django ofrece una forma sencilla de manejar el enrutamiento para el inicio de sesión (`login`) y cierre de sesión (`logout`). Esto se realiza mediante vistas predefinidas que gestionan la sesión del usuario.

### Configuración en `settings.py`: `LOGIN_REDIRECT_URL`, `LOGOUT_REDIRECT_URL`

En el archivo `settings.py`, puedes configurar las URL de redirección después de iniciar sesión o cerrar sesión. El parámetro `LOGIN_REDIRECT_URL` se utiliza para redirigir al usuario a una página específica después de iniciar sesión, y `LOGOUT_REDIRECT_URL` define la URL a la que se enviará al usuario después de cerrar sesión.

### Autorización y permisos: ¿En qué consiste el modelo de permisos de Django?

Django proporciona un sistema de permisos a nivel de aplicación y modelo. Cada usuario puede tener permisos que controlan qué acciones puede realizar (como agregar, modificar o eliminar objetos). Estos permisos se asignan automáticamente a los usuarios cuando se crean, pero también pueden ser personalizados y asignados manualmente.

### ¿Qué son los Mixins?

Los Mixins son clases reutilizables que proporcionan funcionalidades específicas a otras clases sin la necesidad de utilizar herencia tradicional. En Django, los Mixins se utilizan para extender el comportamiento de las vistas, como la autenticación o autorización.

### ¿Para qué nos sirven los Mixins en el modelo Auth?

Los Mixins permiten agregar funcionalidades comunes a las vistas de manera eficiente, como la verificación de si un usuario está autenticado o si tiene ciertos permisos. Los Mixins como `LoginRequiredMixin` y `PermissionRequiredMixin` se usan para restringir el acceso a determinadas vistas basadas en la autenticación o los permisos del usuario.

#### Aplicando `LoginRequiredMixin`

El `LoginRequiredMixin` es un Mixin que garantiza que una vista solo sea accesible para usuarios autenticados. Si un usuario no está autenticado, se le redirige a la página de inicio de sesión.

#### Aplicando `PermissionRequiredMixin`



El `PermissionRequiredMixin` permite restringir el acceso a una vista según los permisos del usuario. Se utiliza para asegurarse de que un usuario tenga un permiso específico antes de permitirle acceder a una vista.

#### **Examinando la tabla auth\_permissions**

La tabla `auth_permissions` en la base de datos almacena todos los permisos de los usuarios en el sistema. Esta tabla contiene los permisos predeterminados y personalizados que pueden ser asignados a los usuarios o grupos.

#### **Redireccionando los accesos no autorizados**

Cuando un usuario intenta acceder a una vista a la que no tiene permiso, Django puede redirigir automáticamente a una página de acceso denegado o mostrar un mensaje de error. Esto se puede configurar para mejorar la experiencia de usuario y mantener la seguridad.

#### **Vistas de autenticación**

Django proporciona vistas predefinidas para gestionar la autenticación de los usuarios, como las vistas de inicio de sesión (`LoginView`), cierre de sesión (`LogoutView`) y cambio de contraseña. Estas vistas pueden ser personalizadas para adaptarse a las necesidades del proyecto.

Última modificación: viernes, 2 de mayo de 2025, 19:58



## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](#). Desarrollado por [Moodle](#)



# Modelo auth de Django

## Objetivos

- Comprender qué es el modelo Auth de Django y cómo maneja la seguridad.
- Aprender cómo utilizar el modelo Auth de Django en un proyecto.
- Realizar migraciones relacionadas con el modelo de autenticación.

## 1. ¿Qué es el modelo Auth de Django?

El modelo Auth de Django es un sistema integrado que se encarga de la autenticación de usuarios. Viene incluido en el paquete `django.contrib.auth` y proporciona las funcionalidades necesarias para manejar usuarios, contraseñas, permisos y grupos en un proyecto Django.

### 1.1. Componentes Principales de Auth

- **User:** Un modelo predefinido que maneja la información básica de los usuarios (nombre, correo electrónico, contraseñas, etc.).
- **Permissions:** Permite asignar permisos específicos a los usuarios o grupos.
- **Groups:** Grupos de usuarios con permisos comunes.
- **Authentication:** La capacidad de autenticar usuarios, es decir, comprobar que las credenciales proporcionadas son correctas.

## 2. ¿Cómo maneja la seguridad Django?

Django maneja la seguridad de varias formas, incluidas la autenticación, la autorización, la protección contra ataques y el manejo de contraseñas de forma segura.

### 2.1. Protección contra ataques comunes

- **CSRF (Cross-Site Request Forgery):** Django incluye una protección integrada contra estos ataques mediante el uso de tokens en los formularios.
- **XSS (Cross-Site Scripting):** Django automáticamente escapa los caracteres en las plantillas para prevenir la inyección de scripts maliciosos.
- **SQL Injection:** Django utiliza ORM (Object-Relational Mapping), lo que ayuda a evitar este tipo de vulnerabilidades al usar consultas parametrizadas.

### 2.2. Manejo de Contraseñas

Django utiliza un sistema de hash de contraseñas fuerte para almacenar las contraseñas de los usuarios de forma segura. No almacena contraseñas en texto claro, sino que las cifra utilizando un algoritmo seguro.

#### Ejemplo de configuración de contraseñas:

En el archivo `settings.py`, Django ofrece configuraciones para personalizar el manejo de contraseñas, como el algoritmo de hash:

```
# settings.py
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
]
```

## 3. ¿Cómo utilizamos el modelo Auth de Django?

Django proporciona el modelo `User` y varias vistas integradas para gestionar la autenticación de usuarios. Puedes utilizar



estas herramientas para manejar el registro, inicio de sesión y cierre de sesión de los usuarios.

### 3.1. Creación de un usuario con el modelo User

Para crear un nuevo usuario, puedes usar el modelo `User` directamente desde el archivo `models.py` o la shell de Django.

#### Ejemplo de creación de usuario:

```
from django.contrib.auth.models import User

# Crear un usuario
user = User.objects.create_user(username='usuario1', password='contraseña_segura')

# Crear un superusuario
superuser = User.objects.create_superuser(username='admin', password='admin123')
```

- `create_user()`: Crea un usuario regular.
- `create_superuser()`: Crea un superusuario, que tiene permisos administrativos.

### 3.2. Autenticación de usuarios

Django ofrece un sistema de autenticación muy sencillo de utilizar, mediante las vistas integradas y el sistema de middleware.

#### Ejemplo de vista para iniciar sesión:

```
from django.contrib.auth import authenticate, login
from django.shortcuts import render, redirect

def login_view(request):
    if request.method == 'POST':
        username = request.POST['username']
        password = request.POST['password']
        user = authenticate(request, username=username, password=password)
        if user is not None:
            login(request, user)
            return redirect('home') # Redirigir a la página principal
        else:
            return render(request, 'login.html', {'error': 'Credenciales inválidas'})
    return render(request, 'login.html')
```

### 3.3. Cierre de sesión

Para cerrar sesión, Django proporciona una vista llamada `logout()`.

#### Ejemplo de vista de cierre de sesión:

```
from django.contrib.auth import logout
from django.shortcuts import redirect

def logout_view(request):
    logout(request)
    return redirect('login')
```

## 4. Ejecutando las Migraciones

Django usa el sistema de migraciones para aplicar los cambios realizados en los modelos de base de datos. Las migraciones se encargan de sincronizar el esquema de la base de datos con los cambios en los modelos.

### 4.1. Creando y ejecutando migraciones

Cuando se hace un cambio en los modelos (por ejemplo, agregar un nuevo campo a un modelo), se deben crear

migraciones y luego aplicarlas a la base de datos.

#### Ejemplo de creación de migración:

```
python manage.py makemigrations
```

Este comando genera archivos de migración que contienen las instrucciones para aplicar los cambios en la base de datos.

#### Ejemplo de ejecución de migraciones:

```
python manage.py migrate
```

Este comando ejecuta las migraciones pendientes y actualiza el esquema de la base de datos.

#### 4.2. Migraciones para el modelo Auth

El modelo `User` y sus tablas relacionadas ya vienen incluidas en las migraciones predeterminadas de Django, por lo que cuando creas un proyecto y ejecutas `migrate`, Django configurará automáticamente la base de datos para que pueda usar el sistema de autenticación.

#### Ejemplo de migración para auth:

```
python manage.py migrate auth
```

Este comando aplicará las migraciones específicas del modelo de autenticación.

Última modificación: viernes, 2 de mayo de 2025, 19:58



## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](#). Desarrollado por [Moodle](#)



# Enrutamiento Login/Logout

## Objetivos

- Comprender cómo funciona el enrutamiento de login/logout en Django.
- Configurar correctamente las URLs y redirecciones para login/logout.
- Entender el modelo de permisos de Django y cómo gestionar la autorización de usuarios.

### 1. Enrutamiento login/logout

En Django, el enrutamiento para las vistas de login y logout se maneja de manera sencilla utilizando las vistas integradas de autenticación y sus respectivas URLs.

#### 1.1. Configuración básica

Django ofrece vistas genéricas para manejar el inicio de sesión y el cierre de sesión de los usuarios de manera predeterminada. Para utilizar estas vistas, debemos configurar las URLs en nuestro proyecto.

##### Ejemplo de enrutamiento para login y logout:

```
# urls.py
from django.contrib.auth import views as auth_views
from django.urls import path

urlpatterns = [
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
]
```

- **LoginView**: Esta vista permite gestionar el inicio de sesión.
- **LogoutView**: Esta vista gestiona el cierre de sesión.

### 2. Configuración en settings.py: LOGIN\_REDIRECT\_URL y LOGOUT\_REDIRECT\_URL

En el archivo `settings.py`, Django ofrece configuraciones específicas para manejar las redirecciones después de que el usuario inicie o cierre sesión. Esto permite personalizar la página a la que será redirigido el usuario después de cada acción.

#### 2.1. LOGIN\_REDIRECT\_URL

La configuración `LOGIN_REDIRECT_URL` se utiliza para especificar a qué página debe ser redirigido el usuario después de iniciar sesión exitosamente.

##### Ejemplo de configuración:

```
# settings.py
LOGIN_REDIRECT_URL = '/home/'
```

En este caso, después de un inicio de sesión exitoso, el usuario será redirigido a la URL `/home/`.

#### 2.2. LOGOUT\_REDIRECT\_URL

De manera similar, `LOGOUT_REDIRECT_URL` define la URL a la que se redirige el usuario después de cerrar sesión.



##### Ejemplo de configuración:

```
# settings.py
LOGOUT_REDIRECT_URL = '/login/'
```



Aquí, después de cerrar sesión, el usuario será redirigido a la página de login.

### 3. Autorización y permisos: ¿En qué consiste el modelo de permisos de Django?

Django tiene un sistema de autorización integrado que permite gestionar los permisos de los usuarios de forma sencilla. El modelo de permisos en Django se utiliza para controlar qué puede y qué no puede hacer cada usuario dentro de una aplicación.

#### 3.1. Permisos en Django

Cada vez que creas un modelo en Django, se generan permisos predeterminados que pueden ser asignados a los usuarios. Django crea permisos como "add", "change", "delete" y "view" para cada modelo.

- **add:** Permite agregar objetos de ese modelo.
- **change:** Permite modificar objetos existentes.
- **delete:** Permite eliminar objetos de ese modelo.
- **view:** Permite ver objetos de ese modelo (en versiones recientes de Django).

#### 3.2. Asignación de permisos a los usuarios

Puedes asignar permisos a los usuarios de dos maneras:

1. **Permisos predeterminados:** Django asigna permisos predeterminados a los usuarios. Por ejemplo, un superusuario tiene todos los permisos por defecto.
2. **Asignación manual:** Puedes asignar permisos específicos a un usuario o grupo. Esto puede hacerse utilizando el administrador de Django o mediante código.

#### Ejemplo de asignación de permisos:

```
from django.contrib.auth.models import User, Permission
from django.contrib.contenttypes.models import ContentType

# Obtener el permiso para cambiar un modelo específico
content_type = ContentType.objects.get_for_model(MyModel)
permission = Permission.objects.get(codename='change_my_model', content_type=content_type)

# Asignar el permiso a un usuario
user = User.objects.get(username='username')
user.user_permissions.add(permission)
```

#### 3.3. Uso de decoradores para permisos

Django proporciona decoradores como `@permission_required` que se pueden usar para restringir el acceso a vistas basadas en los permisos del usuario.

#### Ejemplo de uso de decorador:

```
from django.contrib.auth.decorators import permission_required

@permission_required('app.change_model', raise_exception=True)
def my_view(request):
    # Esta vista solo se puede acceder si el usuario tiene el permiso 'change_model'
    return render(request, 'my_template.html')
```

Última modificación: viernes, 2 de mayo de 2025, 19:58





## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](#). Desarrollado por [Moodle](#)

^

?

## Tabla de permisos

### Objetivos

- Comprender el funcionamiento de la tabla `auth_permissions` en Django.
- Aprender a redirigir accesos no autorizados de manera efectiva.
- Explorar cómo Django maneja las vistas de autenticación.

### 1. Examinando la tabla `auth_permissions`

La tabla `auth_permissions` de Django almacena todos los permisos disponibles en el sistema para los modelos definidos en las aplicaciones. Django crea automáticamente esta tabla cuando se usa el sistema de autenticación y autorización, y cada vez que se crea un nuevo modelo con permisos, estos se agregan a esta tabla.

Puedes consultar esta tabla para ver qué permisos están definidos en el sistema y qué usuarios o grupos tienen acceso a estos permisos.

#### Consultando los permisos:

```
from django.contrib.auth.models import Permission
permissions = Permission.objects.all()
for permiso in permissions:
    print(permiso.name, permiso.codename)
```

Este código imprime todos los permisos disponibles en el sistema, junto con su nombre y código.

### 2. Redireccionando los accesos no autorizados

En Django, puedes redirigir a los usuarios no autorizados a una página de inicio de sesión o a una página de error si intentan acceder a una vista protegida. Esto es útil cuando deseas asegurar que solo los usuarios autenticados o con los permisos adecuados puedan acceder a ciertas vistas.

#### Redirigir cuando el usuario no está autenticado:

En la configuración de `settings.py`, puedes definir una URL para redirigir a los usuarios no autenticados:

```
# settings.py
LOGIN_URL = '/login/'
```

Además, cuando utilices `LoginRequiredMixin` en tus vistas, los usuarios no autenticados serán automáticamente redirigidos a la URL definida en `LOGIN_URL`.

#### Ejemplo de redirección personalizada:

```
from django.shortcuts import redirect
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import TemplateView

class MiVista(LoginRequiredMixin, TemplateView):
    template_name = 'mi_template.html'

    def handle_no_permission(self):
        return redirect('pagina_de_error') # Redirige si no tiene permiso
```

Este código redirige al usuario a una página de error personalizada si no tiene acceso a la vista.



### 3. Vistas de autenticación

Django viene con vistas de autenticación predeterminadas que permiten gestionar el inicio de sesión, cierre de sesión y cambio de contraseña. Estas vistas se pueden utilizar directamente o personalizar según las necesidades de tu proyecto.

#### Ejemplo de vista de inicio de sesión:

```
# urls.py
from django.urls import path
from django.contrib.auth import views as auth_views

urlpatterns = [
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
]
```

En este ejemplo, se han añadido las rutas para las vistas de inicio de sesión y cierre de sesión de Django.

#### Vista de inicio de sesión personalizada:

Si necesitas personalizar la vista de inicio de sesión, puedes hacerlo creando tu propia vista en lugar de usar la vista por defecto. Aquí un ejemplo de cómo hacerlo:

```
from django.contrib.auth.forms import AuthenticationForm
from django.shortcuts import render, redirect
from django.contrib.auth import login

def custom_login(request):
    if request.method == 'POST':
        form = AuthenticationForm(request, data=request.POST)
        if form.is_valid():
            user = form.get_user()
            login(request, user)
            return redirect('home') # Redirige al usuario después de iniciar sesión
    else:
        form = AuthenticationForm()

    return render(request, 'login.html', {'form': form})
```

En este caso, se crea una vista de inicio de sesión personalizada que valida los datos del formulario y, si son correctos, inicia sesión al usuario.

Última modificación: viernes, 2 de mayo de 2025, 19:59



## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)



## Ejercicio individual

### Contexto del Proyecto: Aplicación de Gestión de Eventos

Imagina que estás desarrollando una **Aplicación de Gestión de Eventos** para una organización que organiza conferencias y talleres. La aplicación tiene diferentes tipos de usuarios con distintos roles:

- **Usuarios Regulares:** Pueden registrarse para eventos y ver los detalles de los próximos eventos.
- **Organizadores:** Pueden crear, editar y eliminar eventos, así como ver la lista de asistentes.
- **Administradores:** Tienen acceso completo a la gestión de eventos y usuarios.

La tarea es implementar la autenticación y autorización en Django para asegurarte de que los usuarios solo puedan realizar las acciones que les corresponden según su rol.

### Requisitos del Proyecto

#### 1. Configuración del Modelo Auth en Django:

- Activar el sistema de autenticación de Django para permitir que los usuarios se registren, inicien sesión y cierren sesión.
- Utilizar el modelo User de Django para gestionar los usuarios y crear grupos (por ejemplo: "organizers", "admins").

#### 2. Redirección después del Login/Logout:

- Configura los parámetros LOGIN\_REDIRECT\_URL y LOGOUT\_REDIRECT\_URL en el archivo settings.py para redirigir a los usuarios a una página de inicio o a una página de login según corresponda.
- Crear una vista de bienvenida (home.html) donde los usuarios verán una lista de los próximos eventos disponibles.
- Asegurarte de que los usuarios no autenticados sean redirigidos al login si intentan acceder a vistas protegidas.

#### 3. Manejo de Seguridad en Django:

- Configura las medidas de seguridad necesarias para que la autenticación y las sesiones sean seguras.
- Implementa LOGIN\_URL para dirigir a los usuarios a la página de login cuando intenten acceder a una vista restringida.

#### 4. Permisos y Autorización de Usuarios:

- Implementa un sistema de permisos donde los **organizadores** solo puedan crear, editar y eliminar eventos, pero no puedan ver o modificar información sensible de los usuarios.
- Los **administradores** deberán tener acceso completo a todas las vistas.
- Los **usuarios regulares** solo deben poder ver eventos, pero no editar ni eliminarlos.

#### 5. Uso de Mixins:

- Aplica LoginRequiredMixin para asegurarte de que solo los usuarios autenticados puedan acceder a las vistas relacionadas con eventos.
- Aplica PermissionRequiredMixin para restringir el acceso a vistas de creación y edición de eventos, permitiéndolo solo a los organizadores y administradores.

#### 6. Tabla de Permisos:

- Examina la tabla auth\_permission en la base de datos de Django para entender cómo Django maneja los permisos asignados a los usuarios.
- Asigna permisos a los diferentes grupos de usuarios (organizador, administrador) para controlar qué acciones pueden realizar.



#### 7. Redirección de Accesos No Autorizados:

- Configura las vistas para que los usuarios que no tienen permisos adecuados sean redirigidos a una página de error o a la página de acceso no autorizado.
- Implementa un mensaje de error apropiado cuando un usuario intente acceder a recursos sin los permisos necesarios.

**8. Prueba de Autenticación y Autorización:**

- Crea y configura formularios para el login y registro de usuarios.
- Verifica que el sistema de autenticación y autorización funcione correctamente al acceder y modificar eventos.

**Entrega**

- Entregar los archivos del proyecto en un archivo zip o en un repositorio de Github.
- Duración: 1 jornada de clases.
- Ejecución: Individual.

## Sumario de calificaciones

Ocultado a los estudiantes	No
Participantes	34
Enviados	0
Pendientes por calificar	0



## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](#). Desarrollado por [Moodle](#)



[Ver](#) [Hacer un envío](#)

## Ejercicio grupal

### Contexto del Proyecto: Plataforma de Gestión de Eventos

Imagina que estás trabajando en equipo para desarrollar una **Plataforma de Gestión de Eventos** en Django. La plataforma permite a los usuarios registrarse y gestionar eventos como conferencias, conciertos y seminarios. Sin embargo, algunos eventos son privados y solo deben ser accesibles por ciertos usuarios. En este ejercicio, tu equipo deberá configurar el sistema de autenticación y autorización utilizando el modelo Auth de Django para controlar el acceso a diferentes tipos de usuarios.

### Tareas del Proyecto

#### 1. Configuración del Modelo Auth de Django:

- Explorar el modelo **Auth de Django** para gestionar la autenticación de los usuarios.
- Configurar el sistema de autenticación en Django para permitir el registro, inicio de sesión y cierre de sesión de los usuarios. Asegúrate de que los usuarios puedan acceder a las vistas de gestión de eventos solo después de iniciar sesión.

#### 2. Enrutamiento para Login/Logout:

- Crear rutas específicas para iniciar sesión (`/login`) y cerrar sesión (`/logout`).
- Configurar las vistas y redirigir a los usuarios a la página correcta después de iniciar sesión o cerrar sesión utilizando `LOGIN_REDIRECT_URL` y `LOGOUT_REDIRECT_URL` en el archivo `settings.py`.

#### 3. Gestión de Roles y Permisos:

- Implementar un sistema de roles en el que existan tres tipos de usuarios: **administradores**, **organizadores de eventos**, y **asistentes**.
  - **Administradores**: Tendrán acceso completo para crear, editar y eliminar eventos.
  - **Organizadores de eventos**: Podrán crear y gestionar eventos específicos, pero no podrán eliminar eventos.
  - **Asistentes**: Podrán ver los eventos a los que están registrados, pero no podrán modificarlos.
- Utilizar el modelo de **permisos de Django** para controlar el acceso a las vistas de creación y edición de eventos.

#### 4. Uso de Mixins en el Modelo Auth:

- Aplicar `LoginRequiredMixin` a las vistas donde se requiere que el usuario esté autenticado para acceder (como la creación y edición de eventos).
- Usar `PermissionRequiredMixin` para restringir las vistas de edición y eliminación de eventos solo a los administradores y organizadores con permisos adecuados.

#### 5. Redirección de Accesos No Autorizados:

- Configurar una vista de acceso denegado que redirija a los usuarios que intenten acceder a eventos sin tener permisos suficientes.
- Mostrar un mensaje de error cuando un usuario intente acceder a una vista restringida (por ejemplo, un evento privado).

#### 6. Manejo de Errores y Mensajes:

- Implementar mensajes de error claros en caso de que un usuario intente realizar una acción no permitida (como editar un evento sin permisos).
- Utilizar el sistema de mensajes de Django (`messages.error`) para mostrar estos errores de manera amigable en las vistas.

#### 7. Ejecutando las Migraciones:

- Ejecutar las migraciones necesarias para aplicar los cambios en el modelo de usuarios y eventos.
- Asegurarse de que las tablas correspondientes en la base de datos estén creadas correctamente y que

los usuarios puedan ser autenticados y autorizados según sus roles.

#### 8. Exploración de la Tabla auth\_permission:

- Explorar la tabla auth\_permission en la base de datos para ver cómo Django gestiona los permisos.
- Asignar correctamente los permisos a los usuarios para asegurar que solo los administradores y organizadores puedan editar eventos.

#### 9. Configuración de Seguridad:

- Configurar settings.py para asegurar la plataforma, como activar el uso de sesiones y HTTPS para la autenticación.

#### Entrega

- Entregar un archivo zip con todos los archivos del proyecto en un repositorio Github
- Duración: 1 jornada de clases.
- Ejecución: Individual.

## Sumario de calificaciones

Ocultado a los estudiantes	No
Participantes	34
Enviados	0
Pendientes por calificar	0



## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](#). Desarrollado por [Moodle](#)



## Aprendizaje esperado 6

**Implementar módulo de administración de usuarios y permisos utilizando el módulo preconstruido provisto por el framework Django**

### El sitio administrativo de Django

Django proporciona una interfaz administrativa lista para usarse, que permite gestionar usuarios, permisos y otros modelos en la base de datos de manera intuitiva. Este sitio administrativo está preconfigurado para ser seguro y fácil de personalizar.

#### Utilizando manage.py: Creando un superusuario

A través de `manage.py`, puedes crear un superusuario que tendrá permisos completos para acceder y administrar el sitio administrativo de Django. Esto se hace ejecutando el comando `createsuperuser`, que pedirá datos como nombre de usuario, correo electrónico y contraseña.

#### Limitando el acceso al sitio administrativo

El acceso al sitio administrativo de Django puede ser restringido a ciertos usuarios mediante configuraciones de permisos y autenticación. Es posible limitar quién puede ver y editar qué dentro del sitio administrativo utilizando el sistema de permisos de Django.

#### Conociendo el sitio de administración de Django

El sitio de administración de Django es una herramienta poderosa que permite gestionar usuarios, grupos, permisos, modelos personalizados y más. A través de una interfaz web, se puede realizar tareas administrativas de manera eficiente sin necesidad de escribir código.

#### Creando usuarios en el sistema

Desde el sitio administrativo, los administradores pueden crear nuevos usuarios, asignarles contraseñas seguras y configurar sus permisos. Estos usuarios pueden ser asignados a grupos o tener permisos específicos para acceder o modificar ciertos datos.

#### Iniciando el servidor de administración

Para acceder al sitio administrativo, es necesario iniciar el servidor de desarrollo de Django con el comando `runserver`. Luego, el sitio puede ser accedido a través de un navegador en la URL correspondiente (por defecto, `http://127.0.0.1:8000/admin/`).

#### Accediendo al sitio administrativo

Una vez que el servidor esté en ejecución, puedes acceder al sitio administrativo utilizando las credenciales del superusuario que hayas creado. Desde ahí podrás gestionar los usuarios, grupos, permisos y otros aspectos de la aplicación.

#### Probando usuarios en el modelo Auth

Para verificar que la autenticación de los usuarios está funcionando correctamente, puedes probar creando usuarios y luego acceder al sitio administrativo o a otras vistas protegidas para asegurarte de que el sistema de permisos y autenticación está configurado correctamente.

#### Manejando errores en el modelo Auth

El modelo Auth puede generar errores como contraseñas incorrectas, intentos de inicio de sesión fallidos, o problemas de permisos. Django proporciona mensajes de error detallados que pueden ayudar a los administradores a identificar y solucionar problemas rápidamente.

#### Manejo de grupos en página administrativa de Django

En el sitio administrativo de Django, los administradores pueden crear y gestionar grupos de usuarios. Los grupos permiten agrupar usuarios con permisos comunes, facilitando la asignación de permisos a múltiples usuarios de manera eficiente.

**Manejo de permisos por usuario en página administración de Django**

Django permite asignar permisos específicos a cada usuario a través de su interfaz administrativa. Los permisos pueden ser asignados a nivel de modelo, vista o acción, y se pueden gestionar fácilmente desde el sitio administrativo.

Última modificación: viernes, 2 de mayo de 2025, 19:59



## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](#). Desarrollado por [Moodle](#)



## El sitio administrativo de Django

### Objetivos

- Comprender el funcionamiento del sitio administrativo de Django.
- Aprender a crear un superusuario y a gestionar el acceso al sitio administrativo.
- Manejar usuarios, grupos y permisos dentro del sitio administrativo.
- Conocer cómo interactuar con el modelo auth en Django desde el sitio administrativo.

### 1. El sitio administrativo de Django

Django proporciona un sitio administrativo preconstruido y personalizable, que facilita la administración de los datos de la aplicación, sin necesidad de crear interfaces de usuario complejas para la gestión.

El **sitio administrativo** de Django permite gestionar modelos, usuarios, permisos y más, a través de una interfaz web.

**Accediendo al sitio administrativo:** Una vez que tienes Django configurado, puedes acceder al sitio administrativo en <http://127.0.0.1:8000/admin/> cuando el servidor esté en funcionamiento.

Para ello, debes iniciar el servidor con:

```
python manage.py runserver
```

### 2. Utilizando manage.py: Creando un superusuario

El superusuario es el usuario administrador principal que tiene acceso completo a todas las funciones del sitio administrativo. Para crear un superusuario, utiliza el siguiente comando en la terminal:

```
python manage.py createsuperuser
```

Este comando te pedirá que ingreses un nombre de usuario, correo electrónico y contraseña para el superusuario. Una vez creado, podrás acceder al sitio administrativo con estos datos.

### 3. Limitando el acceso al sitio administrativo

A veces es necesario restringir el acceso al sitio administrativo de Django. Puedes hacerlo de varias maneras:

- **Requeriendo autenticación:** Si deseas proteger el sitio administrativo, asegúrate de que el sitio de administración solo sea accesible a los usuarios autenticados. Para hacerlo, puedes agregar el siguiente middleware en `settings.py`:

```
MIDDLEWARE = [
    # Otras configuraciones...
    'django.contrib.auth.middleware.AuthenticationMiddleware',
]
```

- **Limitando el acceso por IP:** En algunos casos, puedes querer que el acceso solo sea permitido desde ciertas direcciones IP. Esto se puede hacer a nivel de servidor (por ejemplo, en la configuración del servidor web) o utilizando middleware personalizado.

### 4. Conociendo el sitio de administración de Django

El **sitio administrativo de Django** es muy poderoso. Te permite gestionar:

- Modelos definidos en tus aplicaciones.
- Usuarios, grupos, y permisos.



- Contenido de la base de datos, como artículos, productos, etc.

Una vez que has creado un superusuario y accedes al sitio en <http://127.0.0.1:8000/admin/>, verás una interfaz de usuario que muestra una lista de los modelos registrados.

Para que un modelo sea accesible en el sitio administrativo, necesitas registrarlo en el archivo `admin.py` de la aplicación correspondiente:

The first screenshot shows a browser window with the URL `127.0.0.1:8000/admin/login/?next=/admin/`. It displays the 'Django administration' login form. The 'Username' field contains 'alfredosalazar' and the 'Password' field contains several dots. A 'Log in' button is at the bottom right. The second screenshot shows the 'Django administration' dashboard. The title bar says 'Django administration'. Below it, 'WELCOME, ALFREDOSALAZAR. VIEW SITE / CHANGE PASSWORD / LOGOUT'. On the left, a sidebar titled 'Site administration' lists 'AUTHENTICATION AND AUTHORIZATION' with 'Groups' and 'Users' entries, each with '+ Add' and 'Change' buttons. On the right, there are two boxes: 'Recent actions' and 'My actions', both of which say 'None available'.

#### Ejemplo de registro de un modelo:

```
# myapp/admin.py
from django.contrib import admin
from .models import MiModelo

admin.site.register(MiModelo)
```

Este código permite que el modelo `MiModelo` sea visible y gestionable desde el sitio administrativo.

#### 5. Creando usuarios en el sistema

Además del superusuario, puedes crear usuarios adicionales desde el sitio administrativo. Los usuarios pueden tener diferentes permisos según lo que necesiten hacer dentro del sistema.

Para crear un nuevo usuario desde el sitio administrativo:

1. Ingresá a la sección de "Usuarios".
2. Haz clic en "Aregar usuario" en la parte superior derecha.
3. Completa los campos requeridos: nombre de usuario, contraseña y otros datos opcionales.

Además, podes asignar permisos y grupos a los usuarios desde la misma pantalla.

## 6. Iniciando el servidor de administración

El servidor de administración de Django se inicia como cualquier otro servidor de desarrollo de Django con el siguiente comando:

```
python manage.py runserver
```

Esto pone en marcha el servidor de desarrollo, y podrás acceder al sitio administrativo a través de <http://127.0.0.1:8000/admin/>.

Recuerda que este servidor solo debe usarse para desarrollo. Para producción, necesitas configurar un servidor adecuado como Apache o Nginx.

## 7. Accediendo al sitio administrativo

Una vez que hayas creado el superusuario y estés ejecutando el servidor con el comando `python manage.py runserver`, puedes acceder al sitio administrativo desde:

```
http://127.0.0.1:8000/admin/
```

Inicia sesión con las credenciales de superusuario que creaste previamente. Desde allí, podrás gestionar los usuarios, grupos, y modelos de la aplicación.

## 8. Probando usuarios en el modelo auth

El modelo `auth` es el que se utiliza para manejar la autenticación y autorización de los usuarios. Puedes probar usuarios en el sitio administrativo gestionando los siguientes elementos:

- **Usuarios:** Crea, edita y elimina usuarios.
- **Grupos:** Asocia usuarios a grupos con permisos predefinidos.
- **Permisos:** Controla qué acciones puede hacer cada usuario, como agregar, editar o eliminar modelos.

Cuando un usuario se loguea en el sitio, el modelo `auth` determina qué acciones están permitidas según los permisos asignados.

## 9. Manejando errores en el modelo auth

Django maneja automáticamente varios errores relacionados con la autenticación, pero puedes personalizarlos para que se ajusten mejor a tu aplicación. Por ejemplo, si un usuario intenta acceder a una vista que requiere autenticación, Django redirigirá al usuario a la página de inicio de sesión.

### Ejemplo de manejo de errores:

```
from django.contrib.auth.decorators import login_required

@login_required
def mi_vista(request):
    return render(request, 'mi_template.html')
```

Si el usuario no está autenticado, Django lo redirigirá automáticamente a la página de inicio de sesión.

## 10. Manejo de grupos en la página administrativa de Django

Los **grupos** en Django son una manera de organizar usuarios con permisos similares. Puedes crear grupos desde el sitio administrativo y asignarles permisos. Luego, puedes agregar usuarios a estos grupos.

**Creando y asignando grupos:**

1. Ve al sitio administrativo de Django.
2. Haz clic en "Grupos" y luego en "Aregar grupo".
3. Asigna permisos al grupo.
4. Luego, puedes asignar a los usuarios a este grupo desde la sección de usuarios.

Esto es útil para la administración de permisos en proyectos grandes, donde los roles de los usuarios son diversos.

**11. Manejo de permisos por usuario en la página administrativa de Django**

Dentro del sitio administrativo, puedes asignar permisos específicos a usuarios individuales. Estos permisos pueden ser generales o específicos para ciertos modelos.

**Ejemplo de asignación de permisos:**

1. Entra a "Usuarios" y selecciona un usuario.
2. En la sección de permisos, marca las casillas de los permisos que deseas asignar al usuario.

**Ejemplo de código para manejar permisos manualmente:**

```
from django.contrib.auth.models import User, Permission

usuario = User.objects.get(username='nombre_usuario')
permiso = Permission.objects.get(codename='add_mimodelo')

usuario.user_permissions.add(permiso)
```

Este código asigna el permiso para agregar instancias del modelo `MiModelo` al usuario `nombre_usuario`

Última modificación: viernes, 2 de mayo de 2025, 19:59



## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](http://LMSACE.com). Desarrollado por [Moodle](http://Moodle)





## Ejercicio individual

### Contexto del Proyecto: Plataforma de Gestión de Productos

Imagina que estás desarrollando una plataforma para gestionar productos en una tienda en línea utilizando Django. El proyecto debe incluir un sistema administrativo donde los usuarios puedan crear, editar y eliminar productos, pero con diferentes niveles de acceso. Tu tarea será configurar y personalizar el sitio administrativo de Django para gestionar productos de manera eficiente.

### Requerimientos

#### 1. Creación de un Superusuario:

- Utilizando `manage.py`, crea un **superusuario** para acceder al sitio administrativo de Django.
- Asegúrate de que el superusuario tenga acceso completo a todas las funcionalidades del sitio administrativo.

#### 2. Accediendo al Sitio Administrativo:

- Inicia el servidor de desarrollo de Django y accede al sitio administrativo en `http://127.0.0.1:8000/admin/`.
- Usa las credenciales del superusuario para iniciar sesión y ver el panel de administración.

#### 3. Creación de un Modelo de Producto:

- Crea un modelo **Producto** en tu aplicación que incluya los campos básicos como `nombre`, `descripción`, `precio`, `stock`, y `fecha de creación`.
- Registra este modelo en el archivo `admin.py` para que sea visible en el sitio administrativo.

#### 4. Manejo de Usuarios:

- Crea varios usuarios de prueba con diferentes permisos (por ejemplo, un usuario administrador y un usuario normal).
- Asigna a estos usuarios distintos niveles de acceso y asegúrate de que solo el administrador pueda eliminar productos.

#### 5. Manejo de Grupos en la Página Administrativa:

- Crea **grupos** en el sitio administrativo de Django (por ejemplo, un grupo de "Administradores" y un grupo de "Gestores de Productos").
- Asigna permisos específicos a cada grupo, como permitir que los administradores puedan editar y eliminar productos, mientras que los gestores de productos solo puedan agregar y modificar productos.

#### 6. Configuración de Permisos para Usuarios:

- En la página administrativa, configura **permisos específicos** para los usuarios. Por ejemplo, asegúrate de que algunos usuarios solo puedan ver los productos, mientras que otros puedan modificar y eliminar productos.
- Prueba que los permisos estén funcionando correctamente al intentar realizar diferentes acciones con usuarios de distintos roles.

#### 7. Limitación de Acceso al Sitio Administrativo:

- Configura tu proyecto para limitar el acceso al sitio administrativo solo a los usuarios autenticados. Asegúrate de que no sea accesible para usuarios no registrados o no autenticados.

#### 8. Manejo de Errores en el Modelo Auth:

- Asegúrate de que el sistema de autenticación maneje correctamente los errores, como contraseñas incorrectas o intentos fallidos de acceso.
- Muestra un mensaje claro de error cuando un usuario intente acceder al sitio administrativo con credenciales incorrectas.

?

#### 9. Probando el Modelo Auth con los Usuarios:

- Realiza pruebas para verificar que los usuarios con diferentes roles tengan acceso solo a las funcionalidades que se les ha permitido.
- Si un usuario sin permisos intenta realizar una acción no permitida (por ejemplo, eliminar un producto), Django debe redirigirlo a una página de error o de acceso denegado.

**Entrega**

- Entregar un archivo zip con todos los archivos del proyecto o un repositorio Github.
- Duración: 1 jornada de clases.
- Ejecución: Individual.

## Sumario de calificaciones

Ocultado a los estudiantes	No
Participantes	34
Enviados	0
Pendientes por calificar	0



## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](#). Desarrollado por [Moodle](#)

?

[Ver](#) [Hacer un envío](#)

## Evaluación del módulo

Imagina que trabajas como desarrollador para una empresa que está creando una pequeña aplicación web para gestionar tareas. El sistema debe permitir a los usuarios autenticarse, gestionar tareas (crear, ver, eliminar) y ver una lista de sus tareas asignadas. Para esta actividad, se manejarán las tareas en memoria, sin conexión a una base de datos real.

### Parte 1: Configuración Inicial

#### 1. Crear el Proyecto Django:

- Crea un proyecto Django llamado **gestor\_tareas** en tu entorno virtual.

#### 2. Crear la Aplicación de Tareas:

- Dentro del proyecto **gestor\_tareas**, crea una aplicación llamada **tareas**.

#### 3. Configura el Proyecto:

- Asegúrate de que la aplicación **tareas** esté registrada en **INSTALLED\_APPS** en el archivo **settings.py**.

#### 4. Configuración de URLs:

- Crea un archivo **urls.py** dentro de la aplicación **tareas** y configura las rutas necesarias para las vistas principales.

### Parte 2: Vistas y Plantillas

#### 1. Crear Vista de Lista de Tareas:

- Crea una vista para mostrar todas las tareas del usuario autenticado.
- Las tareas se manejarán en memoria, usando una lista de diccionarios que almacene el título y la descripción de cada tarea.
- La vista debe mostrar la lista de tareas en una página utilizando una plantilla HTML.

#### 2. Crear Vista para Detalles de Tarea:

- Crea una vista para mostrar los detalles de una tarea individual (solo la tarea seleccionada por el usuario).
- Usa parámetros en la URL para mostrar los detalles de cada tarea.

#### 3. Crear Vista para Agregar Tarea:

- Crea una vista que permita a los usuarios agregar una nueva tarea.
- El formulario debe ser creado con Django Forms. Usa **forms.Form** para recolectar el título y la descripción de la tarea.
- Al agregar una tarea, se añadirá a la lista en memoria.

#### 4. Crear Vista para Eliminar Tarea:

- Crea una vista que permita a los usuarios eliminar una tarea existente.
- Utiliza la URL para identificar la tarea a eliminar y eliminarla de la lista en memoria.

#### 5. Plantillas:

- Utiliza Bootstrap para diseñar las plantillas y asegurarte de que la interfaz sea responsive.
- Crea plantillas para las vistas: lista de tareas, detalle de tarea, agregar tarea y eliminar tarea.

### Parte 3: Autenticación y Seguridad

#### 1. Autenticación de Usuarios:

- Implementa vistas para que los usuarios puedan **registrarse**, **iniciar sesión** y **cerrar sesión**.
- Utiliza el sistema de autenticación de Django (**django.contrib.auth**).
- Asegúrate de que los usuarios no autenticados no puedan acceder a la vista de lista de tareas. Usa el



decorador `login_required` para proteger las vistas.

## 2. Protección de Vistas:

- Asegúrate de que los usuarios solo puedan gestionar (agregar, eliminar, ver) las tareas que han creado.
- La lista de tareas y las vistas de detalle deben ser privadas para cada usuario autenticado.

## Parte 4: Despliegue y Pruebas

### 1. Pruebas de Funcionalidad:

- Realiza pruebas para asegurarte de que las vistas y formularios funcionen correctamente. Verifica lo siguiente:
  - Las tareas se muestran correctamente en la vista de lista.
  - Los usuarios pueden agregar y eliminar tareas.
  - El sistema de autenticación funciona correctamente (registro, inicio de sesión, cierre de sesión).
  - Las tareas solo se pueden ver, agregar o eliminar por el usuario que las creó.

### 2. Configuración de Servidor de Producción:

- Configura la aplicación para ejecutarse en un entorno de producción, asegurándote de que se utilicen configuraciones adecuadas de `ALLOWED_HOSTS`, `DEBUG`, etc.

## Parte 5: Entrega

### 1. Documentación:

- Redacta una breve documentación explicando cómo se estructura tu proyecto y las funcionalidades principales que implementaste.
- Incluye instrucciones sobre cómo ejecutar el proyecto, configurar el entorno virtual y ejecutar las migraciones.

### 2. Entrega:

- Entrega el código en un repositorio de GitHub o similar, con el `README` correctamente configurado.

## Sumario de calificaciones

Ocultado a los estudiantes	No
Participantes	34
Enviados	0
Pendientes por calificar	0



Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

?

Copyright © 2017 -Desarrollado por [LMSACE.com](http://LMSACE.com). Desarrollado por [Moodle](http://Moodle)

^

?

[Ver](#) [Hacer un envío](#)

## Evaluación de portafolio

### Instrucciones

En función de tu proyecto personal previamente establecido, deberás implementar clase a clase las diferentes tecnologías y competencias técnicas adquiridas a lo largo del curso.

Recuerda que este proyecto irá directamente al registro de evidencia de tu portafolio, el cual deberá demostrar el dominio, competencias técnicas y diferentes habilidades relacionadas con el desarrollo de aplicaciones web utilizando el framework Django.

### Requerimientos Funcionales Mínimos Esperados

- Describir las características fundamentales del framework Django para el desarrollo de aplicaciones empresariales acorde al entorno Python.
- Realizar una investigación sobre las principales características de Django, sus ventajas para el desarrollo de aplicaciones empresariales y cómo facilita el desarrollo rápido y escalable en el entorno Python.  
**Ejemplo:** Comparación de Django con otros frameworks para aplicaciones empresariales.
- Utilizar las herramientas administrativas provistas por el framework para la configuración de un nuevo proyecto web Django.
- Configurar un nuevo proyecto web en Django utilizando las herramientas de administración del framework, como `django-admin startproject` y `django-admin startapp`.  
**Ejemplo:** Creación de un proyecto básico y configuración de su estructura.
- Implementar una aplicación web Django utilizando templates para el despliegue de páginas con contenido dinámico que dan solución a un requerimiento.
- Diseñar una interfaz web dinámica utilizando los sistemas de plantillas de Django para mostrar información proveniente de una base de datos.  
**Ejemplo:** Crear una página que muestre una lista de productos o usuarios, con datos provenientes de un modelo de base de datos.
- Implementar formularios en un aplicativo web utilizando el framework Django para la captura y procesamiento de información dando solución a un problema.
- Crear formularios web utilizando los formularios de Django para recibir datos del usuario, validarlos y almacenarlos en la base de datos.  
**Ejemplo:** Implementar un formulario para registrar nuevos usuarios o productos en una aplicación.
- Implementar mecanismos de autenticación y autorización para el establecimiento de controles de seguridad utilizando el framework Django.
- Configurar el sistema de autenticación de Django para permitir la creación de usuarios, inicio de sesión, y control de accesos en la aplicación.  
**Ejemplo:** Restringir el acceso a ciertas vistas o páginas según el rol del usuario (administrador, usuario regular, etc.).
- Implementar módulo de administración de usuarios y permisos utilizando el módulo preconstruido provisto por el framework Django.
- Configurar y personalizar el módulo de administración de Django para gestionar usuarios, roles y permisos dentro de la aplicación web.  
**Ejemplo:** Crear una vista de administración que permita gestionar los permisos de usuarios, sus datos y otros aspectos de la seguridad.

### Entrega

- **Repositorio en GitHub:**
  - Subir todos los proyectos y configuraciones relacionadas con el desarrollo de la aplicación web utilizando Django.
  - Mantener un historial de cambios mediante commits con mensajes descriptivos.
  - Incluir un archivo README.md que explique la estructura y funcionamiento de la aplicación desarrollada.?

## Sumario de calificaciones

Ocultado a los estudiantes	No
Participantes	34
Enviados	0
Pendientes por calificar	0



## Contacta

✉ Correo electrónico : [contacto@skillnest.com](mailto:contacto@skillnest.com)

Copyright © 2017 -Desarrollado por [LMSACE.com](#). Desarrollado por [Moodle](#)

