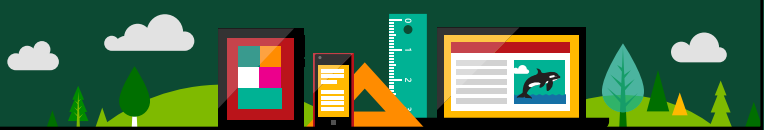

1

## Agenda

| | |
|---|---|
| 1 ¿Qué es Django? | 2 Rutas, Vistas y Plantillas |
| 3 Bases de Datos | 4 Formularios |
| 5 Gestión de Usuarios | 6 API Rest |



4

# ¿Qué es Django?




5

## ¿Qué es Django?

**Django es un framework de desarrollo web de código abierto escrito en Python**. Su propósito principal es facilitar la creación de aplicaciones web seguras, escalables y de alto rendimiento mediante un enfoque basado en la reutilización de código y la simplicidad.

Fue **creado por Adrian Holovaty y Simon Willison en 2003**, mientras trabajaban en un periódico digital.

La primera versión pública, Django 0.90, fue lanzada en julio de 2005. Desde entonces, Django ha evolucionado significativamente, manteniéndose como uno de los frameworks más populares.

También **se pueden crear APIs REST utilizando Django REST Framework (DRF)**. DRF es una biblioteca poderosa que extiende Django para facilitar el desarrollo de APIs escalables y seguras.

django REST framework

6

# Visual Studio Code como IDE para Django

**Visual Studio Code (VS Code)**
Editor de código abierto, es una de las mejores opciones para desarrollar proyectos con Django debido a su ligereza, extensibilidad y compatibilidad con Python.

Es gratuito, multiplataforma y cuenta con múltiples extensiones que mejoran la experiencia de desarrollo. Para mejorar la experiencia en Django, instalaremos las siguientes extensiones en VS Code:

| Extensión | Función |
|---|---|
| **Python** (Microsoft) | Soporte oficial para Python (necesario para ejecutar código). |
| **Django** | Resaltado de sintaxis y fragmentos de código para Django. |
| **Django Template** | Soporte para archivos `.html` con sintaxis de Django (`{% %}` y `{{ }}`). |
| **SQLite Viewer** | Para visualizar la base de datos SQLite directamente en VS Code. |
| **REST Client** | Permite probar APIs directamente desde VS Code sin usar Postman. |

7

# Instalación de Python

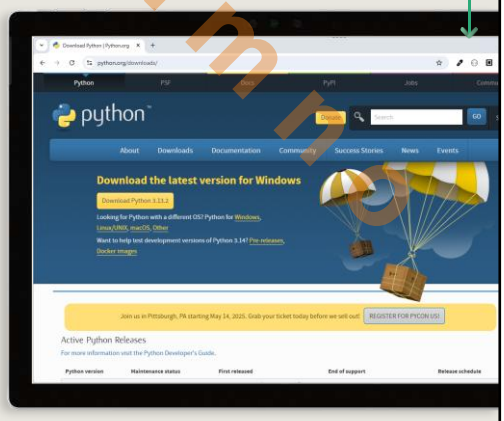
1. Descargar Python desde la página oficial:
2. Instalar Python asegurándote de marcar la opción "*Add Python to PATH*" antes de hacer clic en ***Install Now***.
3. Verificar la instalación abriendo una terminal y ejecutando:

```sh
python --version
python3 --version
```

https://www.python.org/downloads/

**Descargar e instalar la última versión de Python para Windows**

Marcarmos la opción "*Add Python to PATH*" antes de hacer clic en **Install Now**.




8

## Instalación de Visual Studio Code

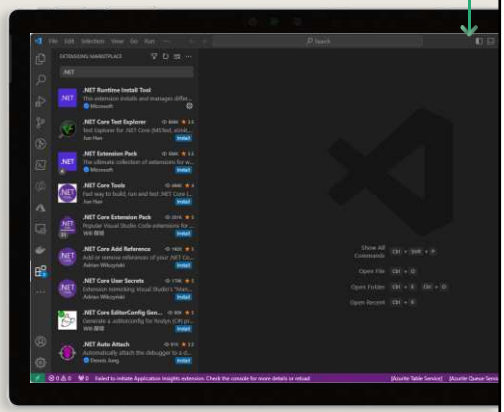
1. Descargamos e instalamos Visual Studio Code.
2. Finalizada la instalación instalamos las extensiones que añaden características al entorno de desarrollo.

Las extensiones de *VSCode* son paquetes de código que se ejecutan dentro del entorno de desarrollo.

https://visualstudio.microsoft.com/es/

**Las extensiones que necesitamos para proyectos Django**

- Python de Microsoft
- Django y Django Template
- SQLite Viewer
- REST Client
- vscode-icons (recomendó)



9

## Crear un Entorno Virtual e instalar Django

Un entorno virtual en Python es una carpeta aislada dentro de tu proyecto que contiene su propia instalación de Python y sus propios paquetes.

Esto evita conflictos entre proyectos que requieren diferentes versiones de las mismas librerías.

1. Abre una terminal y ve a la carpeta de tu proyecto.
2. Crea el entorno virtual con.
3. Activar el entorno virtual.
4. Instalar paquetes dentro del entorno virtual, como Django.
5. Al finalizar, desactivar el entorno virtual

```sh
cd ruta/del/Proyecto
python -m venv venv
Cd
pip install Django
django-admin --version
deactivate
```

10

## Trabajar con un Entorno Virtual

Cada vez que trabajes en tu proyecto, recuerda activar el entorno virtual antes de instalar o ejecutar cualquier paquete.

**Ver las librerías instaladas en el entorno virtual**

- Puedes ver los paquetes que tienes instalados con:
  **`pip list`**
- Si quieres guardar las dependencias en un archivo para compartirlas con otros, usa:
  **`pip freeze > requirements.txt`**
- Para instalarlas en otro entorno:
  **`pip install -r requirements.txt`**

11

## Crear un Proyecto Django

El comando **`django-admin`** es una herramienta de línea de comandos que Django proporciona para gestionar proyectos y aplicaciones de manera eficiente.

Permite crear proyectos, iniciar aplicaciones, ejecutar servidores de desarrollo y mucho más.

1. Dentro de la carpeta de tu proyecto, ejecuta.
2. Entra en la carpeta del proyecto.
3. Ejecuta el servidor de desarrollo para comprobar que funciona.
4. Abre en el navegador en http://127.0.0.1:8000/ y deberías ver la página predeterminada de Django.

```sh
django-admin startproject mi_Proyecto

cd mi_proyecto

python manage.py runserver
```

12

## ¿Qué es el servidor de desarrollo de Django?

El **servidor de desarrollo de Django es una herramienta incluida en el framework de trabajo** de Django, que esta diseñado para simplificar el proceso de desarrollo y pruebas de las aplicaciones web durante la fase de desarrollo.

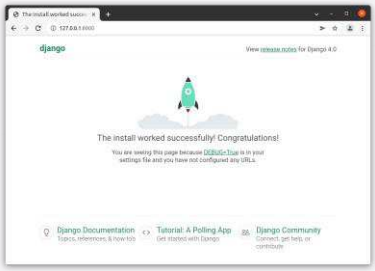
Es un **servidor web ligero que se ejecuta en tu máquina local** y se utiliza durante el desarrollo de una aplicación en Django. No está pensado para ser utilizado en producción, ya que no es tan robusto ni eficiente como otros servidores web, como Apache o Nginx.

**¿Cómo se utiliza?**

```sh
python manage.py runserver

python manage.py runserver 8080

python manage.py runserver 0.0.0.0:8000
```




13

## Definición de Proyecto y Aplicación

**Proyecto**

- Un proyecto es el conjunto completo de configuraciones, archivos y aplicaciones que trabajan juntas para formar una solución web.
- Contiene configuraciones globales (en **`settings.py`**), enrutamiento principal (**`urls.py`**) y otros archivos esenciales.
- Un proyecto puede contener múltiples aplicaciones.

**Aplicaciones**

- Una aplicación es un módulo independiente dentro del proyecto que maneja una funcionalidad específica.
- Puede ser reutilizable en otros proyectos Django.
- Se registra en el archivo de configuración (**`settings.py`**) dentro de la lista **`INSTALLED_APPS`**.



14

## Componentes de una Aplicación en Django

Una aplicación Django típicamente incluye los siguientes archivos y componentes:

**Modelos (`models.py`)**

- Define la estructura de la base de datos utilizando clases de Python.
- Cada clase representa una tabla en la base de datos.

**Vistas (`views.py`)**

- Contiene la lógica de negocio y define cómo se manejan las solicitudes y respuestas.
- Puede devolver respuestas en HTML, JSON, etc.

**URLs (`urls.py`)**

- Define las rutas o *endpoints* de la aplicación.
- Se conecta con el enrutador principal del proyecto.

15

## Componentes de una Aplicación en Django

**Formularios (`forms.py`) (Opcional, pero común)**

- Maneja la validación y procesamiento de formularios en Django.

**Plantillas (`templates/`)**

- Contiene archivos HTML que se renderizan con datos del backend.

**Archivos estáticos (`static/`)**

- Imágenes, archivos CSS y JavaScript utilizados en la aplicación.

**Serializadores (`serializers.py`) (en caso de APIs con Django REST Framework)**

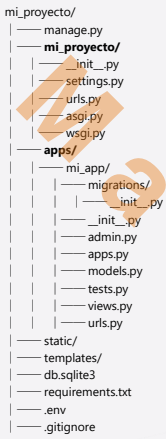
- Convierte los datos entre modelos y formatos JSON/XML.

**Pruebas (`tests.py`)**

- Permite realizar pruebas automatizadas para verificar el funcionamiento de la aplicación

16

## Estructura de un proyecto




**Raíz del Proyecto**

- **`manage.py`**, script para ejecutar comandos de Django (migraciones, creación de apps, etc.).

**Carpeta Principal del Proyecto (`mi_proyecto/`)**

- **`settings.py`**, configuración del proyecto (bases de datos, apps instaladas, etc.).
- **`urls.py`**, enrutamiento principal del proyecto.

**Aplicaciones (`apps/`)**

- cada aplicación dentro del proyecto tiene su propia estructura.

**Carpeta de Recursos**

- `static/` archivos estáticos (CSS, JavaScript, imágenes, etc.).
- `templates/` plantillas HTML para renderizar vistas.



17

## Archivo settings.py

El archivo **`settings.py`** es uno de los componentes más importantes de un proyecto Django.

Contiene la configuración central de tu proyecto, donde se definen parámetros esenciales que afectan cómo se comporta el proyecto en diferentes entornos.

- **SECRET_KEY:** Es una cadena secreta utilizada para la criptografía en Django, como la creación de tokens o sesiones. No debe compartirse ni ser puesta en repositorios públicos.
- **DEBUG:** Controla si el proyecto está en modo de desarrollo (*True*) o en producción (*False*). En producción, debe ser *False* para mejorar la seguridad.
- **ALLOWED_HOSTS:** Define qué dominios o direcciones IP pueden acceder a tu proyecto Django. En producción, siempre se debe especificar.
- **STATIC_URL:** Es la URL donde se pueden acceder los archivos estáticos.
- **MEDIA_URL:** Es la URL base para los archivos subidos por los usuarios.
- **STATICFILES_DIRS:** Rutas adicionales donde Django buscará archivos estáticos.

18

## Archivo settings.py

- **MEDIA_ROOT y STATIC_ROOT:** Rutas en el sistema de archivos donde se almacenarán los archivos estáticos y multimedia.
- **LANGUAGE_CODE:** Define el idioma predeterminado para la aplicación.
- **TIME_ZONE:** Define la zona horaria predeterminada.
- **USE_I18N:** Activa la internacionalización (traducción de la aplicación a varios idiomas).
- **USE_L10N:** Controla si se utiliza el formato de localización para fechas, números, etc.
- **CSRF_COOKIE_SECURE:** Asegura que la cookie CSRF se transmita solo a través de HTTPS.
- **SESSION_COOKIE_SECURE:** Similar a la anterior, para las sesiones.
- **SECURE_SSL_REDIRECT:** Forza el redireccionamiento de HTTP a HTTPS en producción.



19

## Archivo settings.py

La sección **INSTALLED_APPS** contiene las aplicaciones de Django que están activadas en el proyecto. Esto incluye tanto las aplicaciones predeterminadas como las personalizadas.

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

20

## Archivo settings.py

La configuración de la base de datos se define en la variable **DATABASES**.

Por defecto, Django está configurado para usar SQLite, pero puedes configurarlo para usar otros sistemas de gestión de bases de datos como PostgreSQL, MySQL, etc.

```python
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'mi_base_de_datos',
        'USER': 'mi_usuario',
        'PASSWORD': 'mi_contraseña',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

21

## Base de datos de Django

La **base de datos es una parte esencial de la arquitectura del framework** y se utiliza para almacenar toda la información de la aplicación web, incluidos los datos de los usuarios y los registros generados a través de las interacciones de los usuarios con la aplicación.

- La base de datos en Django **tiene como finalidad almacenar los datos de la aplicación**, como registros de usuarios, datos de formularios, artículos, comentarios, transacciones, entre otros. Además, gestiona la persistencia de datos, lo que permite que la información sea guardada y recuperada entre diferentes sesiones de usuario.
- La base de datos **es obligatoria para la mayoría de las aplicaciones Django**, ya que este framework está diseñado para interactuar con bases de datos relacionales de manera muy eficiente. En general, el uso de una base de datos es una de las características clave de Django.
- **Los usuarios del panel de administración (Admins) son registrados en la base de datos**. Django incluye un sistema de autenticación de usuarios por defecto que maneja los usuarios, contraseñas y permisos, y todos esos datos se guardan en la base de datos.

Para usar el sistema de administración de Django (admin), debes crear superusuarios o usuarios con privilegios de administrador ejecutando el comando: **`python manage.py createsuperuser`**

22

## Archivo settings.py

El **middleware son capas intermedias que procesan las solicitudes entrantes y salientes**.

Django incluye un conjunto de middlewares predeterminados, pero puedes agregar más o eliminar algunos si es necesario.

```python
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

23

## Archivo settings.py

Django proporciona un sistema de envío de correos electrónicos. Puedes configurarlo de esta manera:

```python
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_PORT = 587
EMAIL_USE_TLS = True
EMAIL_HOST_USER = 'mi_correo@gmail.com'
EMAIL_HOST_PASSWORD = 'mi_contraseña'
```



24

## Comandos útiles para gestionar un proyecto Django

- `django-admin startproject mi_proyecto` → Crea un nuevo proyecto.
- `django-admin help <comando>` → Consultar la ayuda de un comando.
- `python manage.py startapp mi_app` → Crea una nueva aplicación.
- `python manage.py runserver` → Inicia el servidor de desarrollo.
- `python manage.py makemigrations` → Crea las migraciones para los modelos.
- `python manage.py migrate` → Aplica las migraciones a la base de datos.
- `python manage.py createsuperuser` → Crea un usuario administrador.
- `python manage.py shell` → Abre la consola interactiva de Django.




25

## Patrón MVC (Modelo-Vista-Controlador)

Los patrones de diseño y patrones arquitectónicos son conceptos fundamentales para la estructuración de aplicaciones de software.

El patrón **MVC (Modelo-Vista-Controlador)** y el patrón **MVT (Modelo-Vista-Template)**, utilizado en Django, son ejemplos claros de cómo estructurar aplicaciones de forma eficiente y manejable.

El patrón MVC es un patrón arquitectónico clásico utilizado en aplicaciones de software para separar las diferentes preocupaciones de la aplicación. Cada componente tiene un rol específico:

- **Modelo**: Representa la estructura de los datos de la aplicación y la lógica de negocio. Responsable de acceder a los datos, a menudo mediante una base de datos, y de devolverlos al controlador o a la vista según sea necesario.
- **Vista**: Es la interfaz de usuario, la parte visual de la aplicación que muestra los datos al usuario. Se encarga de la presentación de los datos proporcionados por el modelo.
- **Controlador**: Es el intermediario entre el modelo y la vista. Recibe las entradas del usuario desde la vista y decide qué acciones realizar sobre los datos en el modelo, para luego actualizar la vista.

26

## Patrón MVT en Django (Modelo-Vista-Template)

En Django, el patrón arquitectónico utilizado es muy similar al MVC, pero con algunas diferencias clave.

El patrón se denomina **MVT (Modelo-Vista-Template)**, y se organiza de la siguiente manera:

- **Modelo**: es la capa de la aplicación que se encarga de la lógica de datos.
  Define la estructura de la base de datos y contiene métodos de acceso y manipulación de los datos. Los modelos en Django son definidos por clases que heredan de **`django.db.models.Model`**.
- **Vista**: no es exactamente lo mismo que en el patrón MVC.
  En lugar de manejar la parte visual (como en MVC), la vista en Django es **responsable de procesar las solicitudes del usuario**, interactuar con el modelo para obtener los datos necesarios, y devolver una respuesta (usualmente una plantilla HTML renderizada). La vista **en Django actúa más como el controlador en MVC**.
- **Template (Plantilla)**: Aquí es donde está la mayor diferencia con el patrón MVC.
  **El Template es lo que se conoce como la "vista" en MVC**. Es la capa que se encarga de mostrar la interfaz de usuario (la presentación de los datos). **Los templates son archivos HTML** que pueden incluir etiquetas especiales de Django para la renderización dinámica de datos.

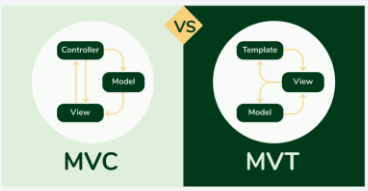


27

## Patrón MVC vs Patrón MVT en Django

Aunque el patrón MVC y el patrón MVT de Django son bastante similares, hay una ligera variación en el nombre y el papel de algunos componentes:

- En MVC, la Vista se encarga de mostrar los datos (UI), mientras que el Controlador maneja la lógica.
- En MVT, la Vista es responsable de manejar la lógica y redirigir el flujo de trabajo. Esto hace que, en Django, la parte de la presentación (la Vista) sea manejada por los Templates.
- Controlador en MVC se traduce en Vista en Django, que tiene la responsabilidad de tomar decisiones sobre qué datos mostrar y cómo mostrarlos.




28

# Rutas, Vistas y Plantillas




29

## Ficheros de URLs

Las URLs de un proyecto o aplicación se definen utilizando el archivo **`urls.py`**.

Este archivo es donde mapeas las vistas a URLs específicas que los usuarios pueden visitar.

Cada aplicación de puede tener su propio archivo que define las rutas para esa aplicación en particular utilizando la función **`path()`** de **`django.urls`**.

En el archivo principal del proyecto, deberá incluir las URLs definidas en las aplicacionesutilizando la función **`include()`** de **`django.urls`**.

```
mi_proyecto/
    ├── blog/
    │   ├── migrations/
    │   ├── __init__.py
    │   ├── admin.py
    │   ├── apps.py
    │   ├── models.py
    │   ├── tests.py
    │   └── urls.py  <-- URLs app
    ├── manage.py
    ├── mi_proyecto/
    │   ├── __init__.py
    │   ├── settings.py
    │   ├── urls.py  <-- URLs proyecto
    │   └── wsgi.py
```

30

# Función path()

La función **`path()`** se utiliza para definir las rutas (*URLs*) en el archivo **`urls.py`**, y toma varios parámetros para configurar la URL y su vista correspondiente.

**Sintaxis de `path()`**

```
path(route, view, kwargs=None, name=None)
```

**Parámetros de `path()`**

- **`route`**: La ruta de la URL.
- **`view`**: La vista que se debe ejecutar cuando se accede a la URL.
- **`kwargs`**: Parámetros adicionales que puedes pasar a la vista, como un diccionario.
- **`name`**: Un nombre opcional para la ruta, utilizado para referenciarla fácilmente en plantillas y vistas.



31

# Función path()

**Descripción de los parámetros**

**`route`** (requiere un tipo de dato str)

- Este parámetro es la cadena de texto que define la ruta de la URL.
- Define la estructura de la URL que el usuario debe ingresar en el navegador.
- Puedes usar parámetros dinámicos dentro de la ruta usando **`<tipo:parametro>`**, como **`<int:id>`** o **`<str:nombre>`**, para capturar valores dinámicos de la URL.

```python
path('productos/', views.productos),  # Ruta simple
path('producto/<int:id>/', views.detalle_producto),  # Ruta con parámetro dinámico
```

32

# Función path()

### Descripción de los parámetros

**`view`** (requiere una vista, que puede ser una función o una clase vista)

- Este parámetro especifica la vista que se debe llamar cuando se accede a la URL.
- La vista puede ser una función basada en vista (FBV) o una clase basada en vista (CBV).
- Si se usa una clase basada en vista (CBV), debes llamarla con **`.as_view()`**.

```python
path('inicio/', views.inicio),  # Función de vista
path('producto/<int:id>/', ProductoDetailView.as_view()),  # Clase basada en vista
```



33

# Función path()

### Descripción de los parámetros

**`kwargs`** (opcional, valor por defecto: **`None`**)

- Este parámetro permite pasar un diccionario de argumentos adicionales a la vista.
- Es útil cuando necesitas enviar datos adicionales a la vista que no están directamente relacionados con los parámetros de la URL.
- En la vista, puedes acceder a estos valores adicionales a través del objeto **`kwargs`**. Sin embargo, generalmente se usan los parámetros dinámicos de la URL para pasar datos a las vistas.

```python
path('producto/<int:id>/', views.detalle_producto, kwargs={'extra_data': 'valor'}),
```

34

# Función path()

```html
<a href="{% url 'detalle_producto' id=producto.id %}">Ver Producto</a>
```

### Descripción de los parámetros

**`name`** (opcional, valor por defecto: **`None`**)

- Este parámetro se utiliza para asignar un nombre a la ruta, lo cual es útil cuando necesitas referenciar la URL en otras partes de la aplicación.
- Por ejemplo, en plantillas con el tag **`{% url %}`** o en el código Python con **`reverse()`**.
- Proporcionar un nombre hace que la URL sea más fácil de mantener, ya que puedes cambiar la estructura de la URL en un solo lugar sin tener que modificar todas las referencias.

```python
path('producto/<int:id>/', views.detalle_producto, name='detalle_producto'),
```

```python
from django.urls import reverse
url = reverse('detalle_producto', kwargs={'id': 1})
```



35

# Función re_path()

La función **`re_path()`** se utiliza para definir rutas utilizando expresiones regulares en lugar de la sintaxis simple de **`path()`**. Esto permite una mayor flexibilidad en la captura de parámetros dentro de las URLs.

```python
re_path(r'patrón_regex', vista, kwargs=None, name=None)
```

### Parámetros de `re_path()`

- **`patrón_regex`** → Expresión regular que define la URL.
- **`vista`** → Función o clase que manejará la solicitud.
- **`kwargs`** (Opcional) → Diccionario de valores adicionales que se pasan a la vista.
- **`name`** (Opcional) → Nombre único para referenciar la URL en plantillas o redirecciones.

36

## Ejemplo de Ruta con re_path()

**Explicación de `r'^saludo/(?P<nombre>\w+)/$'^`**

- **`saludo/`** → Parte estática de la URL.
- **`(?P<nombre>\w+)`** → Captura una palabra y la almacena como nombre.
- **`$`** → Fin de la URL.

**Ejemplo de acceso:**

- **`/saludo/Juan/`** → "Hola, Juan!"
- **`/saludo/Maria/`** → "Hola, Maria!"

Con **`?`** después de **`\w+`** hace que el parámetro nombre sea opcional.

```
re_path(r'^saludo/(?P<nombre>\w+)?/$', saludo)
```

```python
from django.http import HttpResponse
from django.urls import re_path

def saludo(request, nombre):
    return HttpResponse(f"Hola, {nombre}!")

urlpatterns = [
    re_path(r'^saludo/(?P<nombre>\w+)/$', saludo),
]
```

37

## Ejemplo de Ruta con re_path()

**Explicación de:**

```
r'^articulo/(?P<categoria>\w+)/(?P<id>\d+)/$'
```

- **`articulo/`** → Parte estática de la URL.
- **`(?P<categoria>\w+)`** → Captura una palabra (ropa, etc.).
- **`(?P<id>\d+)`** → Captura un número entero (ID del artículo).

**Ejemplo de acceso:**

- **`/articulo/ropa/42/`** → categoria="ropa", id=42.

```python
from django.http import HttpResponse
from django.urls import re_path

def saludo(request, nombre):
    return HttpResponse(f"Hola, {nombre}!")

urlpatterns = [
    re_path(r'^articulo/(?P<categoria>\w+)/(?P<id>\d+)/$',
    views.detalle_articulo),
]
```

38

## Función include()

La función **`include()`** se usa en el archivo **`urls.py`** para incluir otras configuraciones de URL desde aplicaciones específicas dentro del proyecto. Especialmente útil con varias aplicaciones y deseamos organizar mejor las rutas.

```python
path('blog/', include('blog.urls')),  # Incluyendo las URLs de la app "blog"
```

### Parámetros de `include()`

- **`module`** (Obligatorio)
  Especifica el módulo de URL que se incluirá.
  Normalmente se pasa como una cadena, ***`'nombre_de_la_app.urls'`***.
- **`namespace`** (Opcional)
  Se usa para nombrar un conjunto de rutas y evitar conflictos cuando varias aplicaciones tienen nombres de vistas similares. Se define en **`app_name`** dentro del archivo **`urls.py`** de la aplicación.

39

## Parámetros en las rutas

Las rutas en **`urls.py`** pueden incluir parámetros dinámicos, que se capturan y se pasan a las vistas.

```python
path('<tipo:parametro>/', views.mi_vista)
```

- **`int`** → Captura un número entero (`<int:id>/ → 42`)
- **`str`** → Captura una cadena sin / (`<str:nombre>/ → "juan"`)
- **`slug`** → Captura cadenas con letras, números y guiones (`<slug:slug>/ → "mi-articulo-2024"`)
- **`uuid`** → Captura UUIDs (`<uuid:uuid>/ → "550e8400-e29b-41d4-a716-446655440000"`)
- **`path`** → Captura una ruta completa incluyendo / (`<path:ruta>/ → "categorias/ropa/camisetas"`)

40

## Parámetros en las vistas

En **vistas basadas en funciones**, los parámetros se reciben como argumentos en la función:

```python
from django.http import HttpResponse

def detalle_articulo(request, id):
    return HttpResponse(f"Artículo con ID {id}")
```

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

urlpatterns = [
    path('articulo/<int:id>/', views.detalle_articulo, name='detalle_articulo'),
]
```



41

## Parámetros en las vistas

En **vistas basadas en clases**, Django usa **`self.kwargs`** para acceder a los parámetros.

```python
from django.http import HttpResponse
from django.views import View

class DetalleArticuloView(View):
    def get(self, request, id):
        return HttpResponse(f"Vista basada en clase: Artículo con ID {id}")
```

También se puede acceder con **self.kwargs['id']**

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

urlpatterns = [
    path('articulo/<int:id>/', DetalleArticuloView.as_view(), name='detalle_articulo_cbv'),
]
```

42

## Parámetros opcionales

Para hacer que un parámetro sea opcional, debemos darle un valor por defecto en la vista.

```python
from django.http import HttpResponse

def saludo(request, nombre="Invitado"):
    return HttpResponse(f"Hola, {nombre}!")
```

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

urlpatterns = [
    path('saludo/', saludo),            # Sin parámetro (Hola, Invitado!)
    path('saludo/<str:nombre>/', saludo),  # Con parámetro (Hola, Juan!)
]
```



43

## Vistas

Una vista es una función o clase que recibe una solicitud HTTP y devuelve una respuesta HTTP, y cumple las siguientes funciones esenciales:

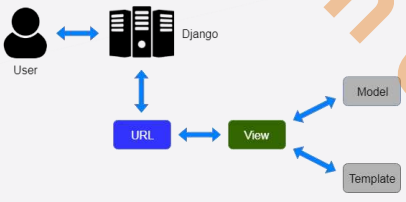
**Recibe una solicitud HTTP**

Una vista es la encargada de recibir la solicitud del cliente (*por ejemplo, una solicitud GET o POST*) enviada a través de una URL que se ha mapeado en el archivo **`urls.py`** del proyecto o de la aplicación.

**Procesa la solicitud**

Una vez que la vista recibe la solicitud, puede realizar diversas tareas:

- Acceder a la base de datos para obtener o modificar datos.
- Realizar lógica de negocio.
- Validar información proveniente de formularios o de la URL.
- Interactuar con otros servicios o librerías.




44

# Vistas II

**Genera una respuesta HTTP**

Después de procesar la solicitud, la vista devuelve una respuesta HTTP. Esta respuesta puede ser de varios tipos:

- **Respuesta simple**, una cadena de texto o una variable que se utiliza para devolver contenido estático.
- **Redirección**, una vista puede redirigir al usuario a una URL diferente.
- **Respuesta con datos dinámicos**, usualmente se utiliza una plantilla HTML (*template*) para renderizar una página dinámica, completando los datos requeridos con información obtenida de la base de datos.

**Composición**

Una vista puede estar compuesta de diversos componentes, como:

- **Consultas a la base de datos**, usando modelos para obtener información.
- **Formularios**, que nos permiten manejar entradas de usuarios.
- **Redirecciones**, que nos permiten cambiar la URL o redirigir a otra vista.



45

# Tipos de vistas

**Vistas basadas en funciones (FBV)**

Son funciones simples que toman una solicitud y devuelven una respuesta.

Son muy directas y fáciles de entender, pero cuando el proyecto crece, pueden volverse difíciles de mantener debido a la repetición de código.

python

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

def mi_vista(request):
    return HttpResponse("¡Hola Mundo!")
```

46

# Tipos de vistas

**Vistas basadas en clases (CBV)**

Son clases que permiten organizar la lógica de la vista en métodos más reutilizables y legibles.

Django proporciona varias clases genéricas que permiten gestionar acciones comunes como mostrar listas de objetos, formularios o detalles de un objeto.

Permiten la reutilización y extensión de funcionalidades mediante herencia, y pueden tener un flujo más estructurado, especialmente en aplicaciones más complejas.

python

```python
from django.http import HttpResponse
from django.views import View

class MiVista(View):
    def get(self, request):
        return HttpResponse("¡Hola Mundo desde una clase!")
```

47

# Tipos de vistas

**Vistas genéricas**

Django ofrece una serie de vistas predefinidas para realizar tareas comunes, como:

- `ListView` para mostrar listas de objetos
- `DetailView` para mostrar detalles de un objeto
- `CreateView`, `UpdateView` y `DeleteView` para gestionar formularios

python

```python
from django.views.generic import ListView
from .models import MiModelo

class MiModeloListView(ListView):
    model = MiModelo
    template_name = 'miapp/mimodelo_list.html'
```

48

## Flujo de trabajo de una vista

**URL dispatcher**

Cuando un usuario accede a una URL, Django usa el archivo **`urls.py`** para determinar qué vista maneja la solicitud.

Las URL en Django se definen mediante patrones que se mapean a vistas específicas.

**Ejecutar la vista**

Una vez determinado qué vista se debe ejecutar, se ejecuta la función o método de la clase correspondiente.

Esta vista procesará la solicitud (por ejemplo, consultando la base de datos o procesando un formulario).

**Respuesta HTTP**

Finalmente, la vista retorna una respuesta, que puede ser un HTML renderizado a partir de un template, un archivo JSON, una redirección o cualquier otro tipo de contenido.



49

## Respuestas de una vista

Las **vistas devuelven respuestas HTTP** (como *HttpResponse*, *JsonResponse*, etc.), pero una de las maneras más comunes de devolver contenido es mediante el uso de plantillas.

**Respuestas HTTP básicas:**

**HttpResponse**: Utilizada para devolver contenido como texto o HTML puro.

```python
from django.http import HttpResponse

def mi_vista(request):
    return HttpResponse("Texto plano o HTML")
```

**JsonResponse**: Devuelve datos en formato JSON, útil para APIs.

```python
from django.http import JsonResponse

def mi_vista(request):
    return JsonResponse({"mensaje": "¡Hola, Mundo!"})
```

50

## Respuestas de una vista

**Respuestas mediante Plantillas:**

Para usar plantillas en Django, primero debes configurar una carpeta de plantillas y luego renderizar la vista con una plantilla. El método render se usa para combinar una plantilla HTML con datos dinámicos (contexto).

El archivo de la vista podría ser algo como:

```python
from django.shortcuts import render

def mi_vista(request):
    contexto = {'nombre': 'Juan'}
    return render(request, 'mi_plantilla.html', contexto)
```

El valor de *nombre* se sustituirá dinámicamente en la plantilla que podría ser algo como:

```html
<html>
    <body>
        <h1>Hola, {{ nombre }}!</h1>
    </body>
</html>
```



51

## Respuestas de una vista

Las vistas también pueden redireccionar a otra URL usando **`redirect`** de **`django.shortcuts`**:

```python
from django.shortcuts import redirect

def mi_vista(request):
    return redirect('https://www.ejemplo.com')
```

Si necesitamos redirigir a una ruta dentro de tu aplicación Django, podemos hacerlo así:

```python
from django.shortcuts import redirect
from django.urls import reverse

def mi_vista(request):
    return redirect(reverse('nombre_de_la_vista'))
```

52

## Vistas basadas en clase con métodos http

Cuando un usuario accede a una URL, Django usa el archivo **`urls.py`** para determinar qué vista maneja la solicitud.

Las URL en Django se definen mediante patrones que se mapean a vistas específicas.

**Ejecutar la vista**

Una vez determinado qué vista se debe ejecutar, se ejecuta la función o método de la clase correspondiente.

Esta vista procesará la solicitud (por ejemplo, consultando la base de datos o procesando un formulario).

**Respuesta HTTP**

Finalmente, la vista retorna una respuesta, que puede ser un HTML renderizado a partir de un template, un archivo JSON, una redirección o cualquier otro tipo de contenido.



53

## Vistas basadas en clase con métodos http

Las vistas basadas en clase (CBV) pueden manejar diferentes métodos HTTP (como *GET*, *POST*, *PUT*, *DELETE*, etc.) implementando métodos específicos dentro de la clase.

Esto te permite gestionar diferentes tipos de solicitudes HTTP en una sola clase de forma ordenada y reutilizable.

Cuando creamos una vista basada en clase, podemos definir métodos para manejar las solicitudes de diferentes tipos.

- **`get(self, request)`**
  Maneja las solicitudes HTTP GET (para obtener datos).
- **`post(self, request)`**
  Maneja las solicitudes HTTP POST (para enviar datos).
- **`put(self, request)`**
  Maneja las solicitudes HTTP PUT (para actualizar datos).
- **`delete(self, request)`**
  Maneja las solicitudes HTTP DELETE (para eliminar datos).


54

# Sobrescribir dispatch()

**`dispatch()`** es el método central de cualquier vista basada en clase, y es responsable de determinar qué método HTTP (como *get*, *post*, etc.) debe ejecutarse.

Si sobrescribes **`dispatch()`**, puedes manejar las solicitudes HTTP de una manera personalizada.

```python
from django.http import HttpResponse
from django.views import View

class MiVista(View):
    def dispatch(self, request, *args, **kwargs):
        if request.method == 'GET':
            return self.mi_get(request, *args, **kwargs)
        # Puedes agregar otros métodos si lo necesitas
        return super().dispatch(request, *args, **kwargs)

    def mi_get(self, request, *args, **kwargs):
        return HttpResponse("Esto responde a una solicitud GET pero no usa el método 'get directamente.")
```



55

# Uso de Decoradores

Una vista recibe una solicitud HTTP (*request*) y devuelve una respuesta HTTP (*response*).

Las vistas manejan la lógica de la aplicación y determinan qué datos se presentan y cómo.

Los **decoradores en se usan para modificar o controlar el comportamiento de las vistas sin cambiar su código** directamente.

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

@login_required
def mi_vista(request):
    return HttpResponse("Solo puedes ver esto si estás autenticado.")
```

| Decorador | Descripción |
|---|---|
| `@login_required` | Restringe el acceso a usuarios autenticados. Redirige a la página de inicio de sesión si el usuario no está autenticado. |
| `@permission_required('app.permiso')` | Verifica si el usuario tiene un permiso específico antes de permitir el acceso a la vista. |
| `@user_passes_test(función)` | Permite definir una función personalizada para evaluar si el usuario tiene acceso a la vista. |
| `@csrf_exempt` | Desactiva la protección CSRF en una vista específica. Se usa en APIs o integraciones externas. |
| `@require_GET` | Restringe la vista para aceptar solo solicitudes HTTP GET. |
| `@require_POST` | Restringe la vista para aceptar solo solicitudes HTTP POST. |
| `@require_http_methods(["GET", "POST"])` | Permite definir qué métodos HTTP están permitidos en la vista. |
| `@never_cache` | Evita que la respuesta de la vista sea almacenada en caché por el navegador o servidores proxy. |
| `@vary_on_headers('User-Agent')` | Modifica el almacenamiento en caché basado en el encabezado especificado. Útil para mostrar contenido diferente según el dispositivo o navegador. |

56

# Jinja

Jinja2 es un **motor de plantillas para Pyt**hon que **permite la generación dinámica de contenido** en aplicaciones web. Se usa ampliamente en frameworks como Flask y Django.

**Mostrar variables en una plantilla**

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

**Estructuras de Control**

```html
{% if edad >= 18 %}
    <p>Eres mayor de edad.</p>
{% else %}
    <p>Eres menor de edad.</p>
{% endif %}
```

```html
<ul>
{% for item in lista %}
    <li>{{ item }}</li>
{% endfor %}
</ul>
```



57

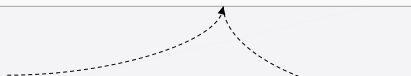
# Jinja

**Uso de Filtros**

Los filtros permiten modificar el contenido de una variable antes de imprimirlo.

```html
{{ mensaje|upper }}  {# Convierte en mayúsculas #}
{{ nombre|capitalize }}  {# Primera letra en mayúscula #}
{{ lista|length }}  {# Devuelve la cantidad de elementos en una lista #}
```

mensaje = "hola mundo"
nombre = "juan"
lista = [1, 2, 3, 4]



HOLA MUNDO
Juan
4

58

Jinja

**Herencia de plantillas (Layouts)**

La herencia de plantillas permite definir una estructura base y que otras plantillas la extiendan.

html (base.html)

```
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}Mi Sitio{% endblock %}</title>
</head>
<body>
    <header>Encabezado</header>
    <main>
        {% block content %}{% endblock %}
    </main>
</body>
</html>
```

html pagina.html (Hereda de base.html)

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

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

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



59

Jinja

**Incluir Fragmentos Reutilizables**

**`{% include %}`** se usa cuando queremos insertar un fragmento de código en varias plantillas sin necesidad de herencia, útil para componentes reutilizables como navbars, sidebars, footers, etc.

html header.html (Fragmento Reutilizable)

```
<header>
    <h1>Mi Sitio Web</h1>
    <nav>
        <ul>
            <li><a href="/">Inicio</a></li>
            <li><a href="/about">Sobre Nosotros</a></li>
        </ul>
    </nav>
</header>
```

html index.html (Incluye header.html)

```
<!DOCTYPE html>
<html>
<head>
    <title>Inicio</title>
</head>
<body>
    {% include "header.html" %}

    <main>
        <h2>Bienvenido a la página principal</h2>
    </main>
</body>
</html>
```

60

Jinja

**extends VS include**

- **`extends`** → Se usa para definir una estructura base y heredarla en otras plantillas.
- **`include`** → Se usa para insertar fragmentos reutilizables en cualquier plantilla.

Puedes usar ambos juntos, por ejemplo:

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

{% block content %}
    {% include "header.html" %}
    <h1>Contenido de la página</h1>
{% endblock %}
```



61

# Ubicación de las plantillas en un proyecto Django

Las plantillas HTML deben colocarse en un directorio especial dentro de la aplicación o en una ubicación global definida en la configuración. Hay dos formas principales de organizar las plantillas:

**Plantillas dentro de una aplicación específica**

Cada aplicación puede tener su propia carpeta **`templates/`**, lo que ayuda a mantener la estructura organizada.

```
mi_proyecto/
├── blog/
│   ├── templates/
│   │   ├── blog/  <-- (Carpeta con nombre de la app)
│   │   │   ├── index.html
│   │   │   ├── detalle.html
│   │   │   ├── base.html
│   ├── views.py
│   ├── models.py
│   ├── urls.py
```

**Plantillas en un directorio global nivel de proyecto**

Podemos colocar todas las plantillas en un directorio global, fuera de las aplicaciones, útil si varias apps comparten las mismas plantillas.

```
mi_proyecto/
├── mi_proyecto/
│   ├── settings.py
│   ├── urls.py
├── templates/  <-- Carpeta global de plantillas
│   ├── base.html
│   ├── home.html
├── blog/
├── tienda/
```

62

# Bases de Datos




63

## Bases de Datos

Django es conocido por su **flexibilidad y facilidad de integración con diferentes motores de bases de datos**, tanto SQL como NoSQL, lo que lo convierte en una herramienta potente y versátil para desarrollar aplicaciones web.

- **Soporte de Bases de Datos SQL**

  Django tiene un soporte robusto para bases de datos relacionales como PostgreSQL, MySQL, SQLite y Oracle.

  Esto se debe a su arquitectura basada en un **ORM (*Object-Relational Mapping*)** que permite trabajar de manera abstracta con bases de datos SQL sin necesidad de escribir consultas SQL complejas manualmente.

- **Soporte de Bases de Datos NoSQL**

  Aunque Django es principalmente conocido por su enfoque en bases de datos SQL, es posible usarlo con ciertos motores NoSQL mediante paquetes de terceros.

  Por ejemplo, Django MongoDB y Django Cassandra permiten integrar bases de datos NoSQL como MongoDB y Cassandra con Django, lo que facilita el trabajo con datos no estructurados o de tipo clave-valor.

  No obstante, la integración de NoSQL en Django no es tan nativa ni tan madura como la de SQL, aunque las comunidades de Django han desarrollado soluciones eficaces.

64

## Trabajar con el ORM integrado de Django


**ORM significa Object-Relational Mapping (Mapeo Objeto-Relacional).**

Es una técnica de programación que permite convertir datos entre sistemas de tipos incompatibles en lenguajes de programación orientados a objetos y bases de datos relacionales.

En términos sencillos, un **ORM permite trabajar con bases de datos utilizando objetos** en lugar de escribir consultas SQL directamente.

- **Mapeo de tablas a clases.** En un sistema de bases de datos relacional, cada tabla es representada como una clase en el lenguaje de programación. Las filas de la tabla corresponden a instancias de la clase, y las columnas de la tabla corresponden a atributos de la clase.
- **Operaciones CRUD.** El ORM proporciona métodos que permiten realizar operaciones CRUD (Crear, Leer, Actualizar y Borrar) sobre la base de datos utilizando objetos de la clase.
- **Abstracción de SQL.** El ORM se encarga de traducir las acciones en el código a las consultas SQL. Esto significa que los desarrolladores no tienen que preocuparse por escribir SQL manualmente, lo que puede ser tedioso y propenso a errores.

65

## Djongo, ORM para MongoDB en Django

djongo

Djongo es una biblioteca que permite usar MongoDB como base de datos en un proyecto Django, manteniendo la interfaz de trabajo de Django ORM.

**Djongo traduce las consultas del ORM de Django en operaciones compatibles con MongoDB**, de manera que el desarrollador pueda seguir trabajando con Django de la misma forma que lo haría con una base de datos relacional.

**Características de Djongo**

- **Integración transparente:** podemos seguir utilizando las características y la sintaxis del ORM de Django, como si estuvieras trabajando con una base de datos SQL, pero con MongoDB en su backend.
- **Modelos de Django:** podemos definir modelos (clases) en Django como de costumbre. Djongo mapea estos modelos a colecciones de MongoDB.
- **Consultas y operaciones CRUD:** permite realizar operaciones CRUD utilizando el ORM de Django, como `Model.objects.filter()`, `Model.objects.create()`, `Model.objects.get()`, etc.
- **Soporte para relaciones:** maneja las relaciones entre modelos (como `ForeignKey`, `ManyToManyField`, etc.), adaptándolas para trabajar con MongoDB.

66

## Conectar SQLite, Microsoft SQL Server, PostgreSQL y MongoDB

El uso de bases de datos es una parte fundamental para el desarrollo de aplicaciones web robustas y escalables.

Gracias a su **ORM (*Object-Relational Mapper*)**, Django facilita en gran medida la interacción con bases de datos relacionales, como PostgreSQL, MySQL o SQL Server, sin necesidad de escribir consultas SQL complejas.

Cuadro resumen con las librerías necesarias para conectar cada base de datos a Django:

| Base de Datos | Librería a instalar | Comando pip |
|---|---|---|
| SQLite | Ninguna (ya viene por defecto) | — |
| PostgreSQL | `psycopg2-binary` | `pip install psycopg2-binary` |
| MS SQL Server | `mssql-django` | `pip install mssql-django` |
| MongoDB (Djongo) | `djongo` | `pip install djongo` |
| MongoDB (Alt.) | `mongoengine` | `pip install mongoengine` |



67

## SQLite (por defecto en Django)

SQLite ya viene integrado con Django y no necesitamos instalar nada extra.


Python (settings.py)

```python
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

68

## PostgreSQL

**Paso 1:** Instalar el conector:

```shell
pip install psycopg2-binary
```

**Paso 2:** Configurar en **`settings.py`**.


```python
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'tu_nombre_db',
        'USER': 'tu_usuario',
        'PASSWORD': 'tu_contraseña',
        'HOST': 'localhost',  # o IP remota
        'PORT': '5432',
    }
}
```

69

## Microsoft SQL Server

**Paso 1:** Instalar el backend de Django para SQL Server:

```shell
pip install mssql-django
```

**Paso 2:** Configura en **`settings.py`**.

**Paso 3:** Agregar en **`INSTALLED_APPS`**:

```python
INSTALLED_APPS = [
    ...
    'mssql',
]
```

```python
DATABASES = {
    'default': {
        'ENGINE': 'mssql',
        'NAME': 'tu_nombre_db',
        'USER': 'tu_usuario',
        'PASSWORD': 'tu_contraseña',
        'HOST': 'localhost',
        'PORT': '1433',
        'OPTIONS': {
            'driver': 'ODBC Driver 17 for SQL Server',
        },
    }
}
```


70

## MongoDB

**Paso 1:** Instalar el conector:

```shell
pip install djongo
```

**Paso 2:** Configurar en **`settings.py`**.


```python
DATABASES = {
    'default': {
        'ENGINE': 'djongo',
        'NAME': 'tu_nombre_db',
        'ENFORCE_SCHEMA': False,
        'CLIENT': {
            'host': 'mongodb://localhost:27017',
        },
    }
}
```



71

## ¿Se pueden usar varias bases a la vez?

Usar múltiples bases de datos en Django es totalmente posible y bastante flexible.

Podemos hacer consultas o guardar en un base de datos específica.

```python
from mi_app.models import Cliente
clientes = Cliente.objects.using('mssql').all()
nuevo_cliente.save(using='mssql')
```

También podemos correr las migraciones para bases de datos específicas.

```shell
python manage.py migrate --database=mssql
```

```python
DATABASES = {
    'default': {...},  # SQLite
    'postgres': {...},
    'mssql': {...},
    'mongo': {...},  # No aplica si usas mongoengine
}
```

72

## Modelo de datos

Una **entidad del modelo de datos es una clase de Python que representa una tabla en la base de datos**. Cada uno de los atributos de la clase representa una columna.

Cuando realizamos cambios el modelo de datos, es necesario crear y aplicar migraciones para reflejar esos cambios en la base de datos.

1. Crear las migraciones:

```shell
python manage.py makemigrations
```

2. Aplicar las migraciones:

```shell
python manage.py migrate
```



73

## Modelo de datos

**Construir el modelo a partir de tablas existentes**

Cuando ya tenemos una base de datos con tablas, podemos generar modelos automáticamente con el comando:

```shell
python manage.py inspectdb > app/models.py
```

**Indicar con qué base de datos trabaja el modelo**

Django permite definir múltiples bases de datos y elegir cuál usar para cada operación.

```python
# Leer desde otra base
clientes = Cliente.objects.using('secundaria').all()

# Guardar en otra base
nuevo = Cliente(nombre="Ana", email="ana@example.com")
nuevo.save(using='secundaria')
```

74

# Modelo de datos

**Forzar el nombre de la tabla al definir el modelo**

Podemos personalizar el nombre de la tabla usando la clase Meta dentro del modelo:

```python
class Cliente(models.Model):
    nombre = models.CharField(max_length=100)

    class Meta:
        db_table = 'clientes_custom'
```

**La clase `Meta`** en los modelos de Django, **es una clase interna que se usa para configurar opciones del modelo** sin afectar directamente los campos. Es muy poderosa y flexible.



75

# ¿Qué es la clase Meta?

La clase `Meta` se define dentro de un modelo y sirve para establecer metadatos, como el nombre de la tabla, permisos, ordenamiento por defecto, ...

```python
class Cliente(models.Model):
    nombre = models.CharField(max_length=100)
    email = models.EmailField()

    class Meta:
        db_table = 'clientes_personalizado'
        ordering = ['nombre']
        verbose_name = 'cliente'
        verbose_name_plural = 'clientes'
```

Con **`managed = False`**, Django no tocará la tabla con migraciones. Útil para modelos que representan vistas SQL o tablas externas.

| Opción | Descripción |
|---|---|
| `db_table` | Nombre exacto de la tabla en la BD. |
| `ordering` | Orden por defecto al consultar (`['campo']` o `['-campo']`). |
| `verbose_name` | Nombre legible para humanos (singular). |
| `verbose_name_plural` | Nombre legible para humanos (plural). |
| `unique_together` | Define restricciones de unicidad entre campos combinados. |
| `index_together` | Define índices combinados. |
| `permissions` | Agrega permisos personalizados. |
| `default_related_name` | Nombre predeterminado para relaciones reversas. |
| `app_label` | Define a qué app pertenece el modelo si no está en el archivo de modelos de esa app. |
| `managed` | Si es `False`, Django no crea ni modifica la tabla con migraciones. Muy útil para tablas existentes. |

76

## Paso 1: Crear Proyecto de Django

Necesitamos tener un proyecto de Django en funcionamiento. Si no lo tenemos, creamos uno ejecutando los siguientes comandos en tu terminal:

```shell
django-admin startproject northwind_project
cd northwind_project
python manage.py startapp customers
```

78

## Paso 2: Definir el modelo Customer

En el archivo **`models.py`**, defimos el modelo de *Customer* de la siguiente manera, emulando algunos de los campos que pueden encontrarse en la tabla *Customers* de *Northwind*:

```python
from django.db import models

class Customer(models.Model):
    customer_id = models.CharField(max_length=5, primary_key=True)  # ID único del cliente
    company_name = models.CharField(max_length=40)  # Nombre de la empresa
    contact_name = models.CharField(max_length=30)  # Nombre de contacto
    contact_title = models.CharField(max_length=30)  # Título de contacto
    address = models.CharField(max_length=60)  # Dirección
    city = models.CharField(max_length=15)  # Ciudad
    region = models.CharField(max_length=15, null=True, blank=True)  # Región
    postal_code = models.CharField(max_length=10)  # Código postal
    country = models.CharField(max_length=15)  # País
    phone = models.CharField(max_length=24)  # Teléfono
    fax = models.CharField(max_length=24, null=True, blank=True)  # Fax
```

79

## Paso 3: Migrar el modelo a la base de datos

En el archivo **`models.py`**, defimos el modelo de *Customer* de la siguiente manera, emulando algunos de los campos que pueden encontrarse en la tabla *Customers* de *Northwind*:

shell

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

80

## Paso 4.1: Crear vistas y operaciones CRUD en Django

**Crear datos (Insertar).** Podemos insertar un nuevo *Customer* utilizando el ORM de Django de la siguiente manera:

python

```python
# Crear un nuevo cliente
new_customer = Customer.objects.create(
    customer_id='ALFKI',
    company_name='Alfreds Futterkiste',
    contact_name='Maria Anders',
    contact_title='Sales Representative',
    address='Obere Str. 57',
    city='Berlin',
    region=None,
    postal_code='12209',
    country='Germany',
    phone='030-0074321',
    fax='030-0076545'
)
```

81

## Paso 4.2: Crear vistas y operaciones CRUD en Django

**Recuperar todos los datos**. Para recuperar todos los datos de los clientes, podemos utilizar el siguiente código:

```python
# Obtener todos los clientes
customers = Customer.objects.all()
for customer in customers:
    print(customer.company_name, customer.city)
```

82

## Paso 4.3: Crear vistas y operaciones CRUD en Django

**Filtrar datos.** Para obtener un solo cliente por la clave primaria usamos **`get()`**. También podemos realizar búsquedas con filtros específicos utilizando el método **`filter()`**. Por ejemplo:

```python
# Obtener un cliente por su ID (customer_id)
try:
    customer = Customer.objects.get(customer_id='ALFKI')
    print(customer.company_name)
except Customer.DoesNotExist:
    print("El cliente no existe")
```

```python
# Filtrar por país
german_customers = Customer.objects.filter(country='Germany')
for customer in german_customers:
    print(customer.company_name, customer.city)

# Filtrar por ciudad
customers_in_berlin = Customer.objects.filter(city='Berlin')
for customer in customers_in_berlin:
    print(customer.company_name)
```

83

## Paso 4.4: Crear vistas y operaciones CRUD en Django

**Actualizar datos.** Podemos actualizar un cliente de la siguiente forma:

```python
# Actualizar el nombre de la empresa de un cliente
customer = Customer.objects.get(customer_id='ALFKI')
customer.company_name = 'Alfreds New Futterkiste'
customer.save()
```

84

## Paso 4.5: Crear vistas y operaciones CRUD en Django

**Eliminar datos.** Para eliminar un cliente:

```python
# Eliminar un cliente por su ID
customer = Customer.objects.get(customer_id='ALFKI')
customer.delete()
```

85

## Paso 4.6: Crear vistas y operaciones CRUD en Django

Vamos a definir algunas posibles vistas en **`views.py`** para hacer el CRUD disponible a través de la web. Usaremos vistas basadas en clases.

```python
# Vista para la lista de clientes
def customer_list(request):
    customers = Customer.objects.all()
    return render(request, 'customers/customer_list.html', {'customers': customers})

# Vista para ver los detalles de un cliente
def customer_detail(request, customer_id):
    customer = get_object_or_404(Customer, customer_id=customer_id)
    return render(request, 'customers/customer_detail.html', {'customer': customer})

# Vista para actualizar un cliente
def customer_update(request, customer_id):
    customer = get_object_or_404(Customer, customer_id=customer_id)
    if request.method == "POST":
        customer.company_name = request.POST['company_name']
        # ... otros campos a actualizar
        customer.save()
        return HttpResponseRedirect(reverse('customer_list'))
    return render(request, 'customers/customer_form.html', {'customer': customer})
```

```python
# Vista para crear un cliente
def customer_create(request):
    if request.method == "POST":
        # Aquí iría la lógica para crear el cliente
        Customer.objects.create(
            customer_id=request.POST['customer_id'],
            company_name=request.POST['company_name'],
            contact_name=request.POST['contact_name'],
            # ... otros campos
        )
        return HttpResponseRedirect(reverse('customer_list'))
    return render(request, 'customers/customer_form.html')

# Vista para eliminar un cliente
def customer_delete(request, customer_id):
    customer = get_object_or_404(Customer, customer_id=customer_id)
    customer.delete()
    return HttpResponseRedirect(reverse('customer_list'))
```



86

## Paso 5: Configurar URLs

En **`urls.py`**, agregaremos las URLs correspondientes para las vistas que acabamos de crear:

python

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

urlpatterns = [
    path('', views.customer_list, name='customer_list'),
    path('create/', views.customer_create, name='customer_create'),
    path('<str:customer_id>/', views.customer_detail, name='customer_detail'),
    path('<str:customer_id>/update/', views.customer_update, name='customer_update'),
    path('<str:customer_id>/delete/', views.customer_delete, name='customer_delete'),
]
```

87

## ¿Qué es `get_object_or_404`?

Es una **función auxiliar de Django** que intenta obtener un objeto de la base de datos y si no lo encuentra, lanza un error 404 (página no encontrada) automáticamente.

| Función | ¿Qué hace? |
|---|---|
| `get_object_or_404` | Devuelve un objeto o lanza 404 si no existe |
| `get_list_or_404` | Devuelve una lista o lanza 404 si está vacía |
| `redirect` | Redirige a otra vista o URL |
| `render` | Renderiza una plantilla HTML |
| `resolve_url` | Resuelve una vista o modelo a una URL |
| `Http404` | Excepción para lanzar error 404 manualmente |



88

## Formularios


89

# Formularios

Los formularios son una forma sencilla y eficiente de gestionar la entrada de datos en una aplicación web.

Django facilita la creación, validación y procesamiento de estos formularios, así como la integración con los modelos y la base de datos.

**Tipos de formularios en Django**

- **Formularios normales (sin modelo)**
  Usados para situaciones donde no se necesita asociar directamente con un modelo de base de datos.
  Se definen usando **`django.forms.Form`**.
- **Formularios basados en modelos (ModelForm)**
  Son formularios que se generan automáticamente a partir de un modelo de base de datos.
  Se definen usando **`django.forms.ModelForm`** y permiten manejar la validación, creación y actualización de registros de la base de datos de forma más eficiente.

90

# Formulario basado en el modelo

Podemos crear un formulario basado en el modelo utilizando **`ModelForm`**.

Esto automáticamente generará los campos del formulario basados en los campos del modelo.

python (forms.py)

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

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

91

# Formularios no asociados al modelo

En Django, puedes crear formularios no asociados a un modelo cuando no necesitas directamente vincular el formulario con un modelo de base de datos, pero aún necesitas recolectar y procesar datos del usuario.

Los formularios no asociados a un modelo se definen utilizando **`django.forms.Form`**.

**Ejemplo: Formulario de contacto**

Muestra cómo crear un formulario de contacto donde un usuario puede enviar su nombre, correo electrónico y mensaje.

1. Definir el formulario en **`forms.py`**
2. Crear la vista en **`views.py`**
3. Mostrar el formulario en un *template*

python (forms.py)

```python
# forms.py
from django import forms

class ContactoForm(forms.Form):
    nombre = forms.CharField(max_length=100, label="Nombre")
    email = forms.EmailField(label="Correo electrónico")
    mensaje = forms.CharField(widget=forms.Textarea, label="Mensaje")
```

python (views.py)

```python
# 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():
            # Aquí puedes hacer lo que necesites con los datos
            nombre = form.cleaned_data['nombre']
            email = form.cleaned_data['email']
            mensaje = form.cleaned_data['mensaje']
            # Puedes agregar un proceso para enviar el mensaje
            # Ejemplo: enviar_correo(nombre, email, mensaje)
            return render(request, 'contacto_exitoso.html')
    else:
        form = ContactoForm()

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

92

# Renderización básica de formularios

La renderización básica de un formulario implica pasar un formulario a un *template* y luego mostrarlo en la interfaz de usuario.

Para esto, se utilizan objetos **`Form`** o **`ModelForm`**.

En el *template* contacto.html, del ejemplo anterior, podemos renderizar el formulario de la siguiente manera.

html (contacto.html)

```html
<!-- contacto.html -->
<h1>Contacto</h1>
<form method="POST">
    {% csrf_token %}
    {{ form.as_p }} <!-- Renderiza el formulario -->
    <button type="submit">Enviar mensaje</button>
</form>
```

93

## Añadir atributos HTML

Al definir formularios podemos añadir atributos HTML, como `class`, `placeholder`, `id`, y otros directamente en los campos del formulario.

**Métodos para añadir atributos HTML**

- Usando widgets para personalizar los atributos

```python
nombre = forms.CharField(
    max_length=100,
    label="Nombre",
    widget=forms.TextInput(attrs={'class': 'form-control', 'placeholder': 'Tu nombre'})
)
```

- Añadiendo los atributos directamente a campos de formularios en el *template*
- Usar `forms.Form` y el atributo `attrs` en un campo específico



94

## Renderización automática

Como vimos antes, para renderizar un formulario en una plantilla HTML, puedes usar la forma básica con los métodos automáticos de renderización, como:

- `{{ form.as_p }}`
- `{{ form.as_table }}`
- `{{ form.as_ul }}`

Estos métodos renderizan el formulario de forma rápida y automática, pero no te dan mucho control sobre el diseño.

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

95

## Renderización manual

Cuando necesitamos un control detallado sobre el formulario, podemos renderizar cada campo individualmente en la plantilla.

**`{{ form.nombre }}`**
Renderiza el campo del formulario (en este caso, nombre) como un campo HTML.

**`{{ form.nombre.errors }}`**
Muestra los errores de validación para ese campo, si los hay.

**`form.nombre.id_for_label`**
Permite que el atributo **`for`** del **`<label>`** apunte al **`id`** del campo en el formulario. Es útil para la accesibilidad y el estilo.

```html
<form method="POST">
    {% csrf_token %}

    <div class="form-group">
        <label for="{{ form.nombre.id_for_label }}">Nombre</label>
        {{ form.nombre }}  <!-- Renderiza el campo "nombre" -->
        {% if form.nombre.errors %}
            <ul class="errorlist">
                {% for error in form.nombre.errors %}
                    <li>{{ error }}</li>
                {% endfor %}
            </ul>
        {% endif %}
    </div>

    <div class="form-group">
        <label for="{{ form.email.id_for_label }}">Correo electrónico</label>
        {{ form.email }}  <!-- Renderiza el campo "email" -->
        {% if form.email.errors %}
            <ul class="errorlist">
                {% for error in form.email.errors %}
                    <li>{{ error }}</li>
                {% endfor %}
            </ul>
        {% endif %}
    </div>

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



## ¿Qué hace form.is_valid()?

El método **`form.is_valid()`** es fundamental en el proceso de validación de formularios.

- **Valida los datos**
  Este método verifica que todos los campos del formulario tengan valores válidos según las reglas que definiste en tu formulario (por ejemplo, si un campo es obligatorio, si un correo electrónico es válido, etc.).
- **Limpieza de datos**
  Durante el proceso de validación, Django también limpia los datos de los campos (por ejemplo, eliminando espacios extra, convirtiendo fechas, etc.) usando el método **`clean()`** de cada campo.
- **Devuelve un booleano**
  **`True`** si todos los campos son válidos y cumplen con las reglas de validación.
  **`False`** si al menos un campo es inválido (y en ese caso, los errores estarán disponibles en el formulario).

## ¿Qué es cleaned_data?

Cuando **`form.is_valid()`** es **`True`**, los datos validados se guardan en el atributo **`cleaned_data`** del formulario.
Este es un diccionario que contiene los valores de los campos con los datos limpios y procesados.

Por ejemplo, si tienes el campo nombre, puedes acceder al valor limpio de la siguiente manera:

```python
nombre = form.cleaned_data['nombre']
```

Esto te garantiza que los datos están validados y procesados (como la conversión de texto a mayúsculas, validación del formato del correo, etc.).

Si un campo es inválido, Django almacena los errores de validación en el formulario, los cuales pueden ser accedidos mediante: **`form.errors`**



98

# Gestión de Usuarios


102

# Sistema de Autenticación

Django viene con un sistema de autenticación incorporado que incluye:

- Gestión de usuarios (registro, login/logout).
- Permisos y grupos.
- Sesiones.
- Middleware para el seguimiento de usuarios autenticados.

El sistema utiliza el modelo **User** por defecto, y todo esto está listo para usar desde el inicio.

python (settings.py)

```python
INSTALLED_APPS = [
    ...
    'django.contrib.auth',
    'django.contrib.contenttypes',
    ...
]
```



103

# Sistema de Autenticación

**Gestión de sesiones y cookies**

Django utiliza cookies para mantener las sesiones activas. Una vez que hacemos login(request, user), Django guarda la sesión del usuario.

python (settings.py)

```python
SESSION_COOKIE_AGE = 1209600  # 2 semanas
SESSION_EXPIRE_AT_BROWSER_CLOSE = False
```

python (cerrar sesión)

```python
from django.contrib.auth import logout

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

104

## Sistema de Autenticación

Necesitamos crear las vistas correspondientes para el **proceso de registro y autenticación** de usuarios:

Python (views.py)

```python
class RegistroForm(forms.ModelForm):
    password = forms.CharField(widget=forms.PasswordInput)

    class Meta:
        model = User
        fields = ['username', 'email', 'password']

def registro(request):
    if request.method == 'POST':
        form = RegistroForm(request.POST)
        if form.is_valid():
            user = form.save(commit=False)
            user.set_password(form.cleaned_data['password'])  # ¡Importante!
            user.save()
            login(request, user)  # Autentica automáticamente tras registrarse
            return redirect('home')
    else:
        form = RegistroForm()
    return render(request, 'registro.html', {'form': form})
```

python (views.py)

```python
from django.contrib.auth import authenticate, login

def iniciar_sesion(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')
        else:
            return render(request, 'login.html', {'error': 'Credenciales inválidas'})
    return render(request, 'login.html')
```

105

## Sistema de Autenticación

**Autorización y permisos de usuario**

Django permite restringir vistas usando decoradores y clases basadas en permisos.

python

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

@login_required
def perfil_usuario(request):
    return render(request, 'perfil.html')
```

python

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

@permission_required('app_name.permiso_especial', raise_exception=True)
def vista_restringida(request):
    ...
```



108

# Sistema de Autenticación

**Personalización del modelo de usuario**

Django permite trabajar con un modelo de usuario personalizado.

python (settings.py)

```python
AUTH_USER_MODEL = 'tu_app.CustomUser'
```

python (models.py)

```python
from django.contrib.auth.models import AbstractUser
from django.db import models

class CustomUser(AbstractUser):
    telefono = models.CharField(max_length=20, blank=True)
```



109

# Sistema de Autenticación

**Implementación de restablecimiento de contraseñas**

Django ya viene con todo listo. Solo necesitas configurar las URLs y plantillas.

python (urls.py)

```python
from django.contrib.auth import views as auth_views

urlpatterns = [
    path('password_reset/', auth_views.PasswordResetView.as_view(), name='password_reset'),
    path('password_reset/done/', auth_views.PasswordResetDoneView.as_view(), name='password_reset_done'),
    path('reset/<uidb64>/<token>/', auth_views.PasswordResetConfirmView.as_view(), name='password_reset_confirm'),
    path('reset/done/', auth_views.PasswordResetCompleteView.as_view(), name='password_reset_complete'),
]
```

python (settings.py)

```python
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'  # Para pruebas
# Para producción, usar SMTP
```

110

# API Rest




121

## ¿Qué es Flask?

Flask es un **micro-framework para Python diseñado para desarrollar aplicaciones web** de forma rápida y sencilla.

A diferencia de frameworks más grandes, no incluye herramientas como ORM, validación de formularios o autenticación por defecto, sino que permite al desarrollador incorporar solo lo que necesita.

- **Ligero y flexible**
- Usa Jinja2 para plantillas
- Soporte para extensiones (SQLAlchemy, Flask-Login, etc.)
- Ideal para APIs REST pequeñas a medianas

Flask

122

## ¿Qué es Django REST Framework (DRF)?

**Django REST Framework (DRF) es una poderosa biblioteca de Django que permite construir APIs** web fácilmente.

Facilita la serialización de datos, control de permisos, autenticación, y más, aprovechando todo el poder de Django.

- Serialización automática de modelos.
- Vistas genéricas y ViewSets para CRUD.
- Soporte para autenticación, permisos, filtros y paginación.
- Navegación web-friendly para probar la API.




123

## Flask vs Django REST Framework

Ambos son poderosos y ampliamente utilizados para desarrollar APIs en Python, pero tienen características distintas que los hacen adecuados para diferentes tipos de proyectos.

| Característica | Flask + Flask-RESTful | Django + DRF |
|---|---|---|
| Enfoque | Minimalista, micro-framework | "Baterías incluidas", completo |
| Configuración | Manual | Mucho viene configurado por defecto |
| ORM | No viene incluido | Usa Django ORM |
| Serialización | Manual o con librerías | DRF incluye Serializers |
| Autenticación | Extensiones (ej: Flask-JWT) | Integrado |
| Ideal para | APIs simples o personalizadas | Aplicaciones grandes y estructuradas |

124

# Instalación de Flask

Flask es muy fácil de instalar, y como es un micro-framework, solo necesitas algunos paquetes básicos para comenzar.

**Flask-SQLAlchemy** facilita la integración de SQLAlchemy con Flask, simplificando la gestión de la base de datos y proporcionándote una solución poderosa para trabajar con bases de datos relacionales sin tener que escribir SQL manualmente.

Puedes verificar si Flask se instaló correctamente con este comando:

***python -m flask --version***

shell
```
pip install flask flask_sqlalchemy
```

125

# Configurar la conexión SQLAlchemy

Flask que utiliza SQLAlchemy como ORM (Object Relational Mapper) para interactuar con una base de datos.

**SQLALCHEMY_DATABASE_URI**
Es una configuración de Flask que indica la URI de la base de datos que se utilizará con SQLAlchemy.

**SQLALCHEMY_TRACK_MODIFICATIONS**
Es una opción de configuración que, cuando se establece en True, hace que *SQLAlchemy* registre los cambios de objetos para poder emitir señales de "modificación".

Establecerla en **`False`** evita que *SQLAlchemy* haga un seguimiento de las modificaciones, lo que ayuda a reducir el consumo de memoria y mejora el rendimiento de la aplicación.

Para conectar con **SQL Server** desde Python, lo más común es usar **pyodbc**.

python
```
from flask import Flask, request, jsonify
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///northwind.db
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

db = SQLAlchemy(app)
```

python
```
from flask import Flask, request, jsonify, abort
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)

# 🔗 Conexión a SQL Server
app.config['SQLALCHEMY_DATABASE_URI'] = (
    "mssql+pyodbc://sa:YourPass123@localhost/Northwind"
    "?driver=ODBC+Driver+17+for+SQL+Server"
)
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

db = SQLAlchemy(app)
```

126

## Modelo de Datos

En Flask, el modelo de datos generalmente se maneja utilizando una base de datos, y se integra comúnmente con SQLAlchemy, que es un ORM (Object-Relational Mapping) para Python.

Esto te permite trabajar con bases de datos usando clases y objetos en lugar de escribir consultas SQL directamente.

python

```python
# 🍊 Modelo Customer (Northwind)
class Customer(db.Model):
    __tablename__ = 'Customers'
    customerid = db.Column(db.String(5), primary_key=True)
    companyname = db.Column(db.String(40), nullable=False)
    contactname = db.Column(db.String(30))
    contacttitle = db.Column(db.String(30))
    address = db.Column(db.String(60))
    city = db.Column(db.String(15))
    region = db.Column(db.String(15))
    postalcode = db.Column(db.String(10))
    country = db.Column(db.String(15))
    phone = db.Column(db.String(24))
    fax = db.Column(db.String(24))
```



127

## Implementando el CRUD

En Flask, un CRUD (Crear, Leer, Actualizar, Eliminar) es una operación básica en aplicaciones web que interactúan con bases de datos.

Usaremos SQLAlchemy para la interacción con la base de datos, ya que facilita el trabajo con bases de datos en aplicaciones Flask.

python

```python
@app.route('/customers', methods=['GET'])
def get_customers():
    customers = Customer.query.all()
    return jsonify([c.__dict__ for c in customers if '_sa_instance_state' not in c.__dict__])

@app.route('/customers/<customer_id>', methods=['GET'])
def get_customer(customer_id):
    customer = Customer.query.get_or_404(customer_id)
    return jsonify({k: v for k, v in customer.__dict__.items() if k != '_sa_instance_state'})
```

python

```python
@app.route('/customers', methods=['POST'])
def create_customer():
    data = request.json
    customer = Customer(**data)
    db.session.add(customer)
    db.session.commit()
    return jsonify({'message': 'Customer created'}), 201

@app.route('/customers/<customer_id>', methods=['PUT'])
def update_customer(customer_id):
    customer = Customer.query.get_or_404(customer_id)
    data = request.json
    for key, value in data.items():
        setattr(customer, key, value)
    db.session.commit()
    return jsonify({'message': 'Customer updated'})

@app.route('/customers/<customer_id>', methods=['DELETE'])
def delete_customer(customer_id):
    customer = Customer.query.get_or_404(customer_id)
    db.session.delete(customer)
    db.session.commit()
    return jsonify({'message': 'Customer deleted'})
```

128

## Middleware de autenticación

En Flask, un middleware es un componente que se ejecuta entre la solicitud y la respuesta.

Para implementar un middleware de autenticación basado en una API Key, puedes crear una función que se ejecute antes de que las vistas sean llamadas.

Esta función verificará si la API Key enviada en las cabeceras de la solicitud es válida.

python

```python
# Middleware para la autenticación con API Key
@app.before_request
def check_api_key():
    api_key = request.headers.get('X-API-KEY')
    if api_key != app.config['API_KEY']:
        abort(401, description="Unauthorized: Invalid API Key")
```



129

## Instalación de Django Rest

Para instalar Django Rest Framework (DRF) en tu proyecto Django, sigue estos pasos:

**Paso 1:** Instalar

**Paso 2:** Instalar Django Rest Framework

Una vez que Django esté instalado, puedes instalar Django Rest Framework.

**Paso 3:** Agregar **`rest_framework`** a **`INSTALLED_APPS`**

Una vez que DRF esté instalado, debes agregar **`'rest_framework'`** a la lista de aplicaciones instaladas en tu proyecto Django.

Abre el archivo settings.py de tu proyecto y busca la sección **`INSTALLED_APPS`**, luego agrega **`'rest_framework'`**.

shell

```
pip install django djangorestframework pyodbc
```

python (settings.py)

```python
INSTALLED_APPS = [
    # otras aplicaciones
    'rest_framework',
]

REST_FRAMEWORK = {
    'DEFAULT_RENDERER_CLASSES': [
        'rest_framework.renderers.JSONRenderer',
    ],
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated',
    ],
}
```

134

## Crear un archivo urls.py

En el archivo **`urls.py`** de tu proyecto o aplicación, debes incluir las rutas de la API.

Por ejemplo, si tienes un modelo llamado Libro, puedes crear una vista API para exponer los datos del modelo.

pyrthon

```python
from django.urls import path, include
from rest_framework import routers
from .views import LibroViewSet

router = routers.DefaultRouter()
router.register(r'libros', LibroViewSet)

urlpatterns = [
    path('api/', include(router.urls)),
]
```



135

## Crear vistas y serializadores

En tu archivo **`views.py`**, puedes usar los *ViewSets* de DRF para exponer tus datos

Django Rest Framework usa *serializadores* para convertir los modelos Django en formatos como JSON.

Crearemos un archivo **`serializers.py`** y define un serializador para tu modelo.

pyrthon (views.py)

```python
from rest_framework import viewsets
from .models import Libro
from .serializers import LibroSerializer

class LibroViewSet(viewsets.ModelViewSet):
    queryset = Libro.objects.all()
    serializer_class = LibroSerializer
```

python (serializers.py)

```python
from rest_framework import serializers
from .models import Libro

class LibroSerializer(serializers.ModelSerializer):
    class Meta:
        model = Libro
        fields = ['id', 'titulo', 'autor', 'fecha_publicacion']
```

136


139


140


Material del Alumno
Coffee Break
Regresamos a las 18.05

141


Lunch Break
1 hora

142


143