



COMILLAS
UNIVERSIDAD PONTIFICIA

ICAI

ICADE

CIHS

Backend: Django

- Desarrollo de aplicaciones y servicios

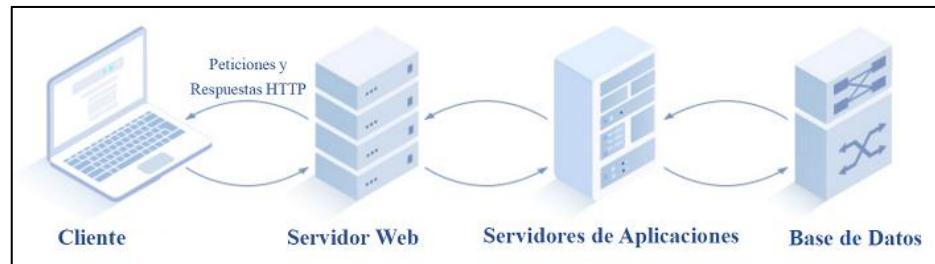
Backend: django

- Introducción
- Tipos de backend
- Backend con python
- Django
- MVT
- MVC
- MVT vs MVC
- ORM
 - ORM en django
- Servidores web en django
- APIREST
- OpenApi
- Swagger



Introducción

El **backend** es la parte del sistema que se encarga de la lógica de negocio, el procesamiento de datos y la comunicación con la base de datos y otras API. Es el "motor" de una aplicación web, funcionando en el servidor y gestionando solicitudes desde el frontend.



El backend generalmente incluye:

- **Servidor:** Maneja las solicitudes del usuario. Ejemplo: Node.js, Apache, Nginx.
- **Aplicación:** Contiene la lógica de negocio. Ejemplo: Express.js, Django, Spring Boot.
- **Base de datos:** Almacena y recupera información. Ejemplo: MySQL, PostgreSQL, MongoDB.

En resumen, el backend garantiza que los datos sean procesados y enviados correctamente al frontend para su visualización.



Introducción

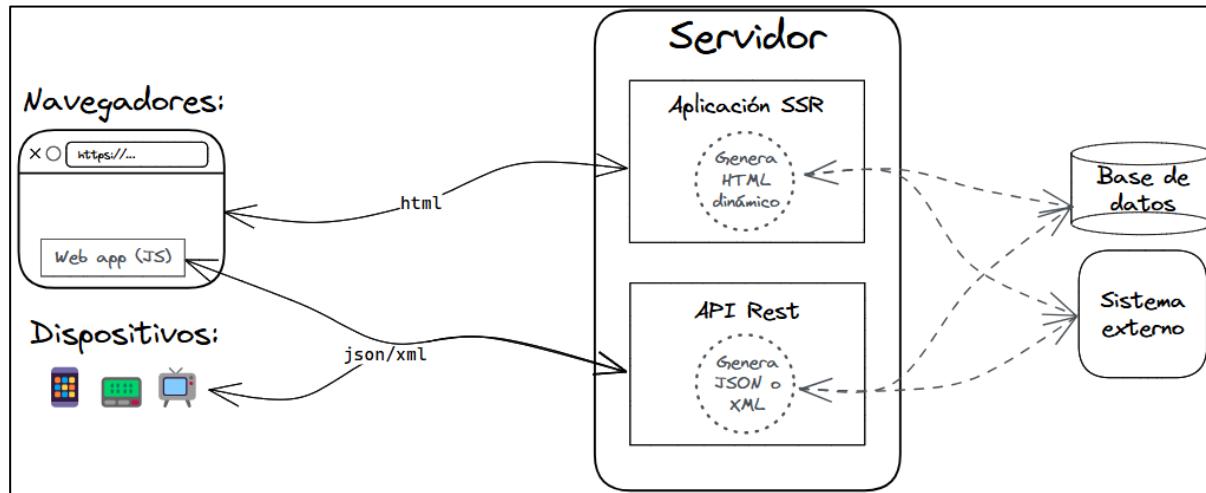




Tipos de backend

Los backends son diferentes dependiendo del protocolo (HTTP, WebSockets, AMQP, etc.), de la funcionalidad (procesamiento, servicio web, etc.) o de sus usuarios (dispositivos IoT, otros servicios, personas, etc.). En el caso de aplicaciones web, existen dos aproximaciones:

- **Server-side rendering (SSR):** la presentación final se genera en el servidor.
- **API REST:** el servidor retorna los datos en un formato independiente que luego será procesado en el cliente para su presentación. En resumen, el backend garantiza que los datos sean procesados y enviados correctamente al frontend para su visualización.



Backend con Python

Ventajas

- Facilidad de lectura y comprensión
- Versatilidad
- Facilidad de integración
- Varios frameworks

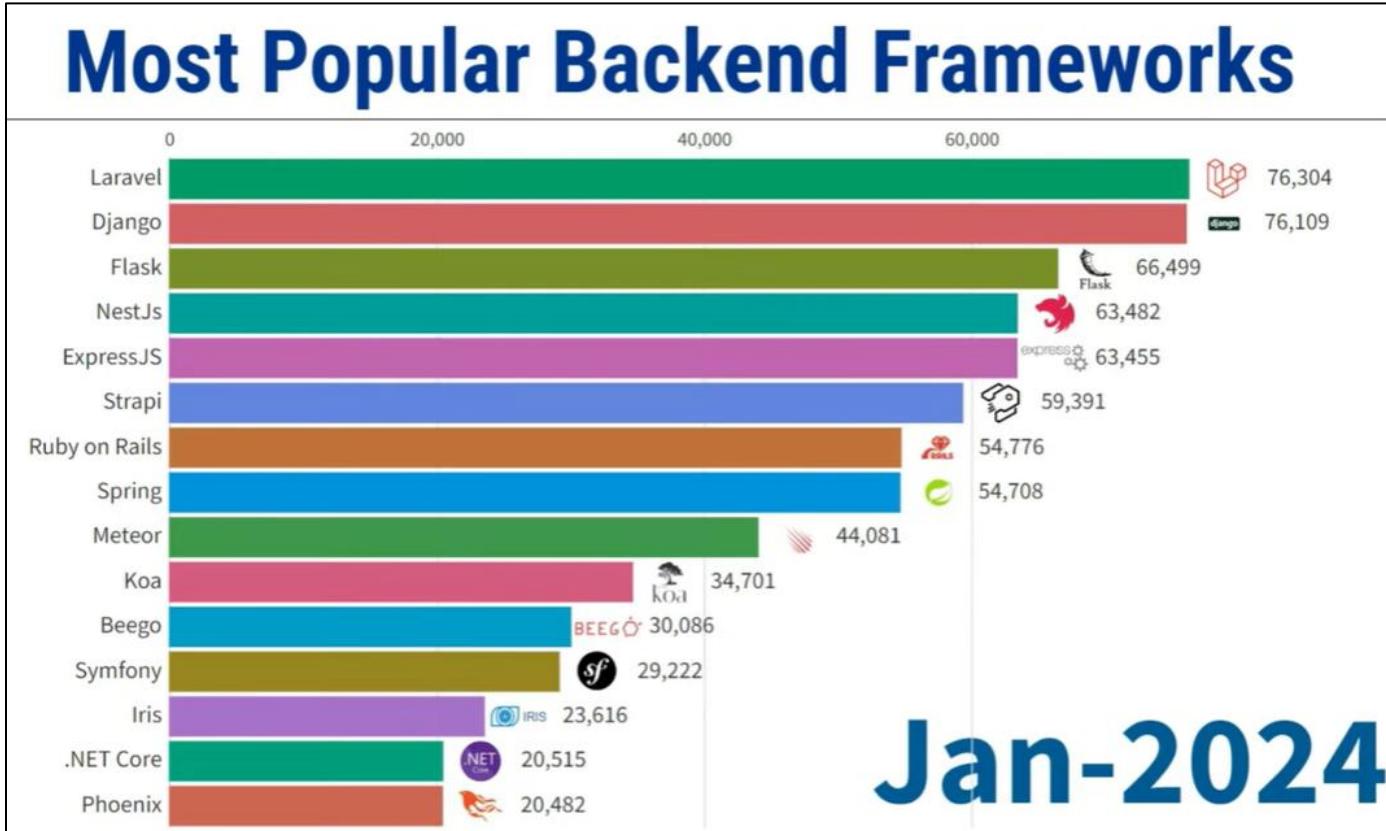


Desventajas

- Limitación en velocidad
- Gran consumo de memoria



Backend con Python



Django

Django es un framework de desarrollo web de código abierto escrito en Python que facilita la creación rápida y eficiente de aplicaciones web robustas y seguras.

Se basa en:

- **Diseño basado en el principio DRY (Don't Repeat Yourself):** Django promueve la reutilización del código y la reducción de la redundancia al proporcionar una arquitectura limpia y modular.
- **Arquitectura MVC (Modelo-Vista-Controlador):** Django sigue este patrón arquitectónico, donde el modelo representa los datos, la vista controla la interfaz de usuario y el controlador gestiona la lógica de la aplicación. Sin embargo, en Django, la vista se refiere a la capa de presentación, y el controlador se maneja internamente por el framework.

Django

- **ORM (Object-Relational Mapping):** Django incluye un ORM que permite a los desarrolladores interactuar con la base de datos utilizando objetos Python en lugar de escribir consultas SQL directamente. Esto simplifica el acceso y la manipulación de datos en la base de datos y hace que el código sea más legible y mantenible.
- **Administrador de Django:** Django proporciona un panel de administración incorporado que facilita la creación de interfaces de administración para gestionar los modelos de datos de la aplicación de manera fácil y rápida.
- **Sistema de plantillas:** Django incluye un sistema de plantillas que permite a los desarrolladores generar contenido dinámico utilizando plantillas HTML con marcadores de posición que son rellenados por datos proporcionados por la aplicación.

Django

- **Seguridad integrada:** Django tiene características de seguridad integradas para ayudar a proteger las aplicaciones web contra ataques comunes, como la inyección de SQL, la protección CSRF (Cross-Site Request Forgery) y la prevención de XSS (Cross-Site Scripting).
- **Escalabilidad y rendimiento:** Django está diseñado para escalar desde pequeñas aplicaciones hasta aplicaciones web de gran escala con un alto volumen de tráfico. Además, proporciona herramientas y prácticas recomendadas para optimizar el rendimiento de las aplicaciones.

Django: ¿por qué?

Open source

Código abierto

Gran comunidad

Respaldada y mantenida



django

Soluciones incorporadas

Formularios, autenticación de usuarios, administración del sitio y seguridad

Popular

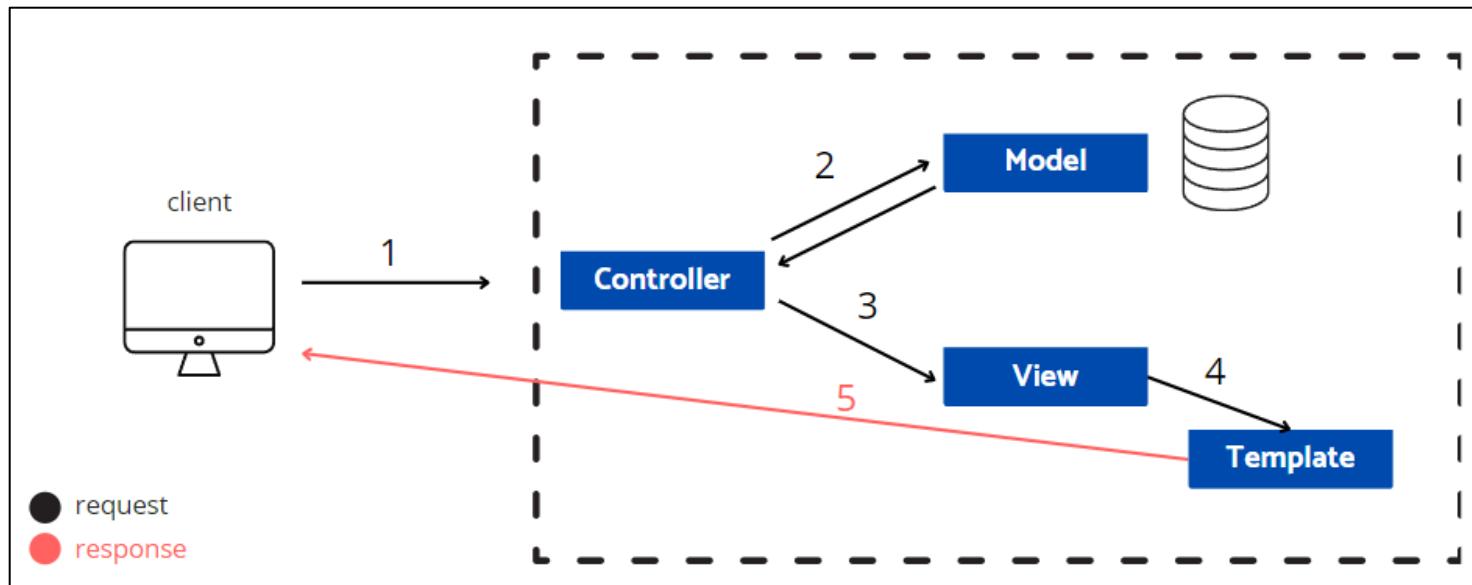
Usado por grandes empresas

Rápido aprendizaje

Muchos tutoriales, abundante documentación y sencillez



MVC



MVC

El patrón de **diseño MVC (Modelo-Vista-Controlador)** es un enfoque arquitectónico utilizado comúnmente en el desarrollo de aplicaciones web y de software en general. Se basa en la separación de la lógica de la aplicación en tres componentes principales: el Modelo, la Vista y el Controlador.

- **Modelo (Model):**

- El modelo representa la capa de acceso a datos de la aplicación.
- Es responsable de gestionar los datos de la aplicación y la lógica del negocio.
- Representa la estructura de los datos, incluyendo su almacenamiento, manipulación y validación.
- Los cambios en el modelo pueden desencadenar actualizaciones en las vistas correspondientes.

- **Vista (View):**

- La vista representa la capa de presentación de la aplicación.
- Es responsable de mostrar los datos del modelo al usuario de una manera adecuada.
- Interactúa con el usuario y muestra la interfaz de usuario, incluyendo todas las interfaces gráficas y elementos visuales.
- Las vistas pueden recibir datos del controlador y mostrarlos al usuario, y también pueden enviar datos al controlador para su procesamiento.

MVC

- **Controlador (Controller):**

- El controlador actúa como intermediario entre el modelo y la vista.
- Es responsable de interpretar las acciones del usuario y actualizar el modelo en consecuencia.
- Gestiona las solicitudes del usuario y coordina la interacción entre el modelo y la vista.
- Los cambios en la vista pueden desencadenar acciones en el controlador, que a su vez pueden actualizar el modelo y/o seleccionar la vista adecuada para mostrar al usuario.

Flujo básico MVC

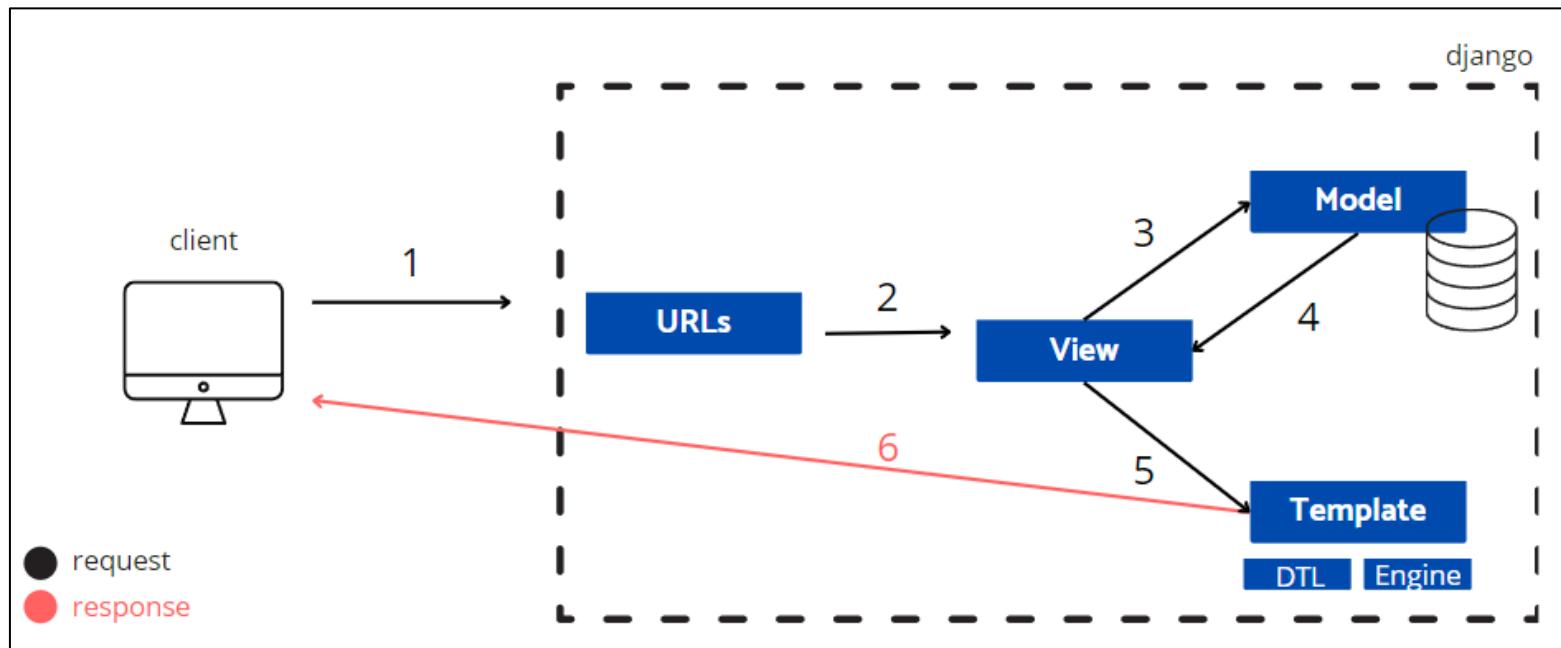
- El usuario hace una petición (Request):** El flujo comienza cuando el usuario realiza una acción en la interfaz de usuario que genera una solicitud al servidor. Por ejemplo, hacer clic en un enlace o enviar un formulario. (*Flecha 1 del diagrama*)
- El controlador recibe la petición (Controller):** El controlador es el primer componente del MVC en recibir la petición del usuario. El controlador interpreta la solicitud del usuario y determina qué acción debe tomar en respuesta a esa solicitud. Basándose en la acción solicitada y otros datos proporcionados por la solicitud (como parámetros de URL o datos de formulario), el controlador puede interactuar con el modelo para recuperar o actualizar datos.
- El controlador interactúa con el modelo (Model):** El controlador puede realizar varias operaciones con el modelo, como recuperar datos de la base de datos, realizar cálculos o actualizar el estado del modelo. El controlador no debe contener lógica de negocio compleja; en su lugar, actúa principalmente como un coordinador entre la vista y el modelo. (*Flecha 2 del diagrama*)
- El modelo procesa la solicitud (Model):** El modelo realiza las operaciones solicitadas por el controlador y procesa la solicitud del usuario. Puede realizar operaciones de lectura y escritura en la base de datos u otras operaciones relacionadas con los datos.

Flujo básico MVC

5. **El controlador selecciona la vista adecuada (Controller):** Despues de que el modelo haya procesado la solicitud, el controlador selecciona la vista adecuada para mostrar al usuario. El controlador puede pasar datos relevantes al modelo a la vista para que sean mostrados al usuario. (*Flecha 3 del diagrama*)
6. **La vista renderiza la respuesta (View):** La vista es responsable de presentar los datos al usuario en una interfaz de usuario adecuada. Utilizando los datos proporcionados por el controlador, la vista renderiza la respuesta y genera la salida final que se enviará al usuario. La salida final puede ser una página HTML, un archivo JSON, un fragmento de XML, etc., dependiendo del tipo de solicitud y del formato de respuesta requerido. (*Flecha 4 del diagrama*)
7. **La respuesta se envía al usuario (Response):** Finalmente, la respuesta generada por la vista se envía de vuelta al navegador del usuario. El usuario recibe la respuesta y puede verla en su navegador como una página web, un archivo descargable, una respuesta de API, etc. (*Flecha 5 del diagrama*)



MVT (Django)



MVT (Django)

El patrón de diseño **MVT (Modelo-Vista-Plantilla)** es una variante del patrón **MVC (Modelo-Vista-Controlador)** adaptado a las características específicas del framework.

- **Modelo (Model):**

- El modelo en Django representa la capa de acceso a datos de la aplicación.
- Define la estructura de los datos y proporciona métodos para interactuar con la base de datos.
- Los modelos de Django se definen mediante clases que heredan de la clase `django.db.models.Model`.
- Cada clase de modelo define un tipo de dato específico y sus relaciones con otros tipos de datos.

Fichero models.py

```
from django.db import models

class Producto(models.Model):
    name = models.CharField(max_length=100, primary_key=True)
```

MVT (Django)

Tipo de Dato	Campo en Django	Ejemplo
Texto corto	models.CharField	nombre = models.CharField(max_length=100)
Texto largo	models.TextField	descripcion = models.TextField(null=True, blank=True)
Números enteros	models.IntegerField	stock = models.IntegerField(default=0)
Números decimales	models.DecimalField	precio = models.DecimalField(max_digits=10, decimal_places=2)
Booleanos	models.BooleanField	disponible = models.BooleanField(default=True)
Fechas	models.DateField	fecha_creacion = models.DateField(auto_now_add=True)
Fecha y Hora	models.DateTimeField	fecha_actualizacion = models.DateTimeField(auto_now=True)
Campos Relacionales	models.ForeignKey (1-N)	categoria = models.ForeignKey(Categoría, on_delete=models.CASCADE)
Campos Relacionales	models.ManyToManyField (N-N)	tags = models.ManyToManyField(Tag)

MVT (Django)

- Vista (View):

- La vista en Django representa la capa de lógica de la aplicación.
- Es responsable de procesar las solicitudes HTTP y devolver las respuestas adecuadas.
- En Django, una vista es una función de Python o una clase basada en funciones que toma una solicitud HTTP y devuelve una respuesta HTTP.
- Las vistas en Django pueden realizar tareas como recuperar datos de la base de datos, procesar formularios, realizar cálculos y renderizar plantillas.

Fichero views.py

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

def tienda_view(request):
    usuario = {"nombre": "Juan"}
    productos = Producto.objects.all()
    contexto = {
        "usuario": usuario,
        "productos": productos
    }
    return render(request, "tienda.html", contexto)
```

MVT (Django)

- Plantilla (Template):

- La plantilla en Django representa la capa de presentación de la aplicación.
- Define la estructura y el diseño de las páginas web que se muestran al usuario.
- **DTL (Django Template Language)** es el lenguaje de plantillas nativo de Django. Se usa para insertar lógica dentro de los archivos HTML, permitiendo mostrar datos dinámicos.
 - Usa dobles llaves ({{ }}) para mostrar variables.
 - Usa etiquetas ({{% %}}) para lógica, como if, for, extends, etc.
 - Tiene filtros (|) para modificar datos (ejemplo: upper, lower, date, etc.).
 - Es seguro (escapa HTML automáticamente para evitar ataques XSS).
- **El Template Engine** es el sistema que Django usa para renderizar plantillas. Soporta varios motores de plantillas, pero los más comunes son DTL y Jinja2.

MVT (Django)

DTL - Fichero en templates/tienda.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Mi Tienda</title>
</head>
<body>
    <h1>Bienvenido, {{ usuario.nombre }}!</h1> {# Muestra una variable #-}

    {% if productos %} {# Condición en DTL #}
        <ul>
            {% for producto in productos %}
                <li>{{ producto.nombre }} - ${{{ producto.precio|floatformat:2 }}}</li> {# Usa filtro floatformat #}
            {% endfor %}
        </ul>
    {% else %}
        <p>No hay productos disponibles.</p>
    {% endif %}
</body>
</html>
```

MVT (Django)

TEMPLATE ENGINEE - Fichero setting.py

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates', #
        Usa DTL
        'DIRS': [BASE_DIR / "templates"], # Directorio donde buscar plantillas
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
            ],
        },
    },
]
```

MVT (Django)

-Urls:

- Aunque no tiene sigla dentro de MVT, desempeña un papel fundamental.
- Se encarga de mapear las URLs de tu aplicación a las vistas (views) correspondientes. Es decir, define qué vista debe ejecutarse cuando un usuario accede a una URL específica en tu sitio web.

Fichero urls.py

```
from django.urls import path
from .views import tienda_view, contacto_view

urlpatterns = [
    path("tienda/", tienda_view, name="tienda"),
    path("contacto/", contacto_view, name="contacto"),
]
```

Flujo básico MVT

- El usuario hace una petición (Request):** El flujo comienza cuando el usuario realiza una acción en la interfaz de usuario que genera una solicitud al servidor, como hacer clic en un enlace o enviar un formulario. (*Flecha 1 del diagrama*)
- La URL maneja la petición:** La primera etapa en el flujo de trabajo de Django es la resolución de la URL. El servidor Django examina la URL solicitada y la compara con un conjunto de patrones de URL definidos en el archivo de configuración de las URL. Cuando se encuentra una coincidencia, se llama a la vista asociada a esa URL. (*Flecha 2 del diagrama*)
- El controlador (vista) recibe la petición:** La vista recibe la solicitud y contiene la lógica para manejarla. La vista interactúa con el modelo para recuperar o actualizar datos según sea necesario. (*Flecha 3 del diagrama*)
- El modelo procesa la solicitud:** Basándose en la solicitud recibida, la vista interactúa con el modelo para realizar operaciones como recuperar datos de la base de datos, realizar cálculos o actualizar el estado del modelo. (*Flecha 4 del diagrama*)

Flujo básico MVT

5. **La vista selecciona la plantilla adecuada:** Después de que el modelo haya procesado la solicitud, la vista selecciona la plantilla adecuada para mostrar al usuario. (*Flecha 5 del diagrama*). La plantilla es responsable de presentar los datos al usuario en una interfaz de usuario adecuada.
6. **La plantilla renderiza la respuesta:** La plantilla renderiza la respuesta utilizando los datos proporcionados por la vista. La plantilla es un archivo HTML que puede contener marcadores de posición para datos dinámicos generados por la vista.
7. **La respuesta se envía al usuario (Response):** Finalmente, la respuesta generada por la plantilla se envía de vuelta al navegador del usuario. El usuario recibe la respuesta y puede verla en su navegador como una página web renderizada. (*Flecha 6 del diagrama*)

¿Por qué usar MVT?

- **Separación de responsabilidades:** El MVT divide claramente las diferentes responsabilidades de una aplicación web en tres componentes distintos: el Modelo, la Vista y la Plantilla. Esto facilita la comprensión y el mantenimiento del código al separar las preocupaciones relacionadas con el acceso a datos, la lógica de la aplicación y la presentación de la interfaz de usuario.
- **Reutilización del código:** Al separar las responsabilidades en diferentes capas, el MVT permite reutilizar el código de manera más eficiente. Por ejemplo, los modelos de datos pueden ser utilizados en múltiples vistas y las plantillas pueden ser compartidas entre diferentes vistas para generar interfaces de usuario coherentes.
- **Flexibilidad y escalabilidad:** El MVT proporciona una estructura flexible que permite escalar y extender fácilmente una aplicación web a medida que crece en tamaño y complejidad. Las vistas y las plantillas pueden ser adaptadas y ampliadas según sea necesario sin afectar el modelo de datos subyacente.

¿Por qué usar MVT?

- **Facilidad de desarrollo y mantenimiento:** El MVT simplifica el desarrollo de aplicaciones web al proporcionar una estructura clara y bien definida. Los desarrolladores pueden enfocarse en implementar la lógica de la aplicación en las vistas y diseñar la interfaz de usuario en las plantillas sin preocuparse por la manipulación directa de los datos en la base de datos.
- **Testing y depuración:** El MVT facilita la escritura de pruebas unitarias y de integración para cada componente de la aplicación por separado. Esto permite probar y depurar de manera efectiva cada aspecto de la aplicación, desde la lógica de negocio hasta la presentación de la interfaz de usuario.

MVC vs MVT

	MVT	MVC
MODELO	Representa <u>la capa de acceso a datos de la aplicación</u> . Define la estructura de los datos y proporciona métodos para interactuar con la base de datos.	También <u>representa la capa de acceso a datos</u> , pero puede incluir lógica de negocio adicional.
VISTA	Representa <u>la capa de lógica de la aplicación</u> . Es responsable de procesar las solicitudes HTTP y devolver las respuestas adecuadas.	Representa <u>la capa de presentación</u> .
PLANTILLA/ CONTROLADOR	Responsable de <u>la presentación de los datos</u> y define la apariencia de las páginas web que se muestran al usuario.	Es el intermediario entre el modelo y la interfaz de usuario (<u>capa de lógica</u>). Responsable de interpretar las acciones del usuario y actualizar el modelo en consecuencia.

ORM

El **mapeo objeto-relacional (ORM)** es una técnica de programación que permite a los desarrolladores trabajar con bases de datos relacionales utilizando objetos de programación en lugar de escribir consultas SQL directamente. En un sistema ORM, los objetos del lenguaje de programación (como clases en Python) se mapean a tablas en una base de datos relacional, y las operaciones sobre estos objetos se traducen automáticamente en consultas SQL que interactúan con la base de datos subyacente.

ORM

- **Abstracción de la base de datos:** Los desarrolladores pueden interactuar con la base de datos utilizando objetos de programación familiares en lugar de tener que escribir consultas SQL directamente. Esto facilita el desarrollo de aplicaciones al ocultar la complejidad de la estructura de la base de datos y las consultas SQL subyacentes.
- **Sintaxis simplificada:** Los ORMs suelen proporcionar una sintaxis más legible y fácil de entender en comparación con el SQL tradicional. Esto puede hacer que el código sea más limpio y mantenible.
- **Portabilidad:** Los ORMs suelen ser independientes del sistema de gestión de bases de datos (DBMS), lo que significa que las aplicaciones pueden ser más portables y pueden cambiar de un DBMS a otro con relativa facilidad.
- **Seguridad:** Los ORMs suelen incluir mecanismos para prevenir ataques de inyección SQL al escapar automáticamente los parámetros de las consultas.
- **Mapeo de relaciones:** Los ORMs pueden manejar relaciones complejas entre tablas de bases de datos y convertirlas en relaciones de objetos en el lenguaje de programación.

ORM en Django

- **Definición de modelos:** En Django, los modelos son clases de Python que representan las tablas en la base de datos. Cada modelo define los campos de la tabla y sus tipos de datos correspondientes. Los modelos se definen en archivos Python dentro de la aplicación Django y heredan de la clase `django.db.models.Model`.
- **Interfaz de consulta:** El ORM de Django proporciona una interfaz de consulta poderosa (`python manage.py shell`) y expresiva para interactuar con la base de datos. Los desarrolladores pueden realizar consultas utilizando métodos de Python encadenados, lo que hace que las consultas sean legibles y fáciles de escribir.
- **Abstracción de la base de datos:** El ORM de Django maneja automáticamente la generación de consultas SQL específicas del motor de base de datos utilizado. Esto significa que los desarrolladores pueden escribir código que sea independiente del DBMS subyacente. Por ejemplo, podemos cambiar de sqlite a postgresql sin tener que hacer ninguna modificación manual de datos.

ORM en Django

- **Migraciones de esquema:** Django incluye una herramienta de migración de esquema que permite a los desarrolladores realizar cambios en el esquema de la base de datos de forma segura y controlada. Las migraciones se definen en archivos Python y se pueden aplicar automáticamente a la base de datos.
 - ***py manage.py makemigrations*** {aplicación}: detecta los cambios que ha habido en los modelos de la aplicación y define una serie de operaciones para trasladar los cambios a la base de datos.
 - ***py manage.py sqlmigrate***: muestra cómo son las operaciones en lenguaje de base de datos que ejecutarán tras la migración del paso anterior.
 - ***py manage.py migrate***: se aplican los cambios del primer paso.
- **Integración con el panel de administración:** El ORM de Django se integra estrechamente con el panel de administración de Django, que proporciona una interfaz de usuario predefinida para interactuar con los datos de la aplicación. Los modelos definidos en la aplicación Django se pueden registrar en el panel de administración con unas pocas líneas de código.

ORM en Django

- **Migraciones de esquema:** Django incluye una herramienta de migración de esquema que permite a los desarrolladores realizar cambios en el esquema de la base de datos de forma segura y controlada. Las migraciones se definen en archivos Python y se pueden aplicar automáticamente a la base de datos.
 - ***py manage.py makemigrations*** {aplicación}: detecta los cambios que ha habido en los modelos de la aplicación y define una serie de operaciones para trasladar los cambios a la base de datos.
 - ***py manage.py sqlmigrate***: muestra cómo son las operaciones en lenguaje de base de datos que ejecutarán tras la migración del paso anterior.
 - ***py manage.py migrate***: se aplican los cambios del primer paso.
- **Integración con el panel de administración:** El ORM de Django se integra estrechamente con el panel de administración de Django, que proporciona una interfaz de usuario predefinida para interactuar con los datos de la aplicación. Los modelos definidos en la aplicación Django se pueden registrar en el panel de administración con unas pocas líneas de código.

Servidor web de Django

Un servidor web es un software o hardware que almacena, procesa y entrega páginas web a los usuarios a través de Internet o una red local. Su función principal es manejar solicitudes HTTP (o HTTPS) de los clientes, como navegadores web, y responder enviando los archivos solicitados (HTML, CSS, JavaScript) o datos en formato JSON o XML.

Es necesario para ejecutar y servir una aplicación web desarrollada en Django. Aunque Django incluye su propio servidor de desarrollo para propósitos de desarrollo y pruebas, para entornos de producción se recomienda usar un servidor web más robusto y escalable. Algunos de los servidores web comúnmente utilizados con Django son:

- **Gunicorn (Green Unicorn):** es un servidor HTTP WSGI (Web Server Gateway Interface) para Python que es ampliamente utilizado con Django en entornos de producción. Es conocido por su velocidad, eficiencia y capacidad para manejar múltiples solicitudes concurrentes. Se puede usar junto con un servidor proxy como Nginx o Apache para manejar las solicitudes HTTP y servir la aplicación Django.
- **uWSGI:** es otro servidor HTTP WSGI popular para Python que se puede utilizar con Django en entornos de producción. Ofrece una amplia gama de características, incluyendo soporte para varios protocolos de red, equilibrio de carga, y gestión avanzada de procesos. Al igual que Gunicorn, uWSGI se puede usar junto con un servidor proxy para servir aplicaciones Django a través de HTTP.

Servidor web de Django

- **Apache HTTP Server:** es uno de los servidores web más populares y ampliamente utilizados en el mundo. Se puede configurar para servir aplicaciones Django utilizando el módulo mod_wsgi, que permite integrar aplicaciones Python con el servidor Apache. Aunque es un servidor web potente y flexible, algunos desarrolladores prefieren servidores como Gunicorn o uWSGI para aplicaciones Django debido a su simplicidad y eficiencia.
- **Nginx:** es un servidor web ligero, de alto rendimiento y escalable que se utiliza comúnmente como servidor proxy inverso o servidor frontal para servidores de aplicaciones web. Se puede usar con servidores como Gunicorn o uWSGI para servir aplicaciones Django y manejar tareas como la compresión de archivos estáticos, el balanceo de carga y el caching. Nginx es especialmente popular en entornos de producción debido a su capacidad para manejar un gran número de solicitudes concurrentes.

APIREST

Una **APIREST (Interfaz de Programación de Aplicaciones Representacional del Estado Transferido)** es un conjunto de reglas y herramientas que permite a dos programas comunicarse entre sí a través de internet de manera eficiente y segura. Esta arquitectura se basa en los principios de REST, que utiliza operaciones HTTP estándar (GET, POST, PUT, DELETE) para realizar acciones sobre recursos, como datos o servicios, de manera uniforme y predecible.



[Clic al vídeo](#)

Principios REST

REST, o Representational State Transfer, es un conjunto de principios de diseño para construir aplicaciones web escalables y fáciles de mantener.

- **Recursos:** En REST, todo es un recurso. En nuestra aplicación de tienda online, los recursos pueden ser cosas como usuarios y productos.
- **Identificadores únicos (URI):** Cada recurso tiene una identificación única, que se representa mediante una URL (Uniform Resource Locator). Por ejemplo:
 - /products para representar la colección de los productos que se venden en la tienda.
 - El recurso se nombra en plural (products en lugar de product).
 - No se deben incluir acciones en las URLs (por ejemplo, en lugar de /deleteProduct/{id}, se debería usar DELETE /products/{id}).
 - Las URLs pueden reflejar la estructura jerárquica de los recursos. Por ejemplo:
 - /users/{id}/posts para representar los posts de un usuario específico.

Principios REST

- **Métodos HTTP:** Utilizamos los métodos HTTP estándar para realizar operaciones en estos recursos:
 - **GET** para recuperar datos. Por ejemplo, **GET /products** para obtener todos los productos (lista).
 - **POST** para crear nuevos recursos. Por ejemplo, **POST /products** para agregar un producto nuevo.
 - **PUT** para actualizar recursos existentes. Por ejemplo, **PUT /products/{product_id}** para actualizar un producto específico. Es una actualización total del producto; es decir, hay que enviar todos los campos.
 - **PATCH** para actualizar algunos campos de recursos existentes. Por ejemplo, **PATCH /products/{product_id}**. Se mandan los campos que se quieren modificar.
 - **DELETE** para eliminar recursos. Por ejemplo, **DELETE /products/{product_id}** para eliminar un producto concreto.
- **Representaciones:** Los datos se intercambian entre el cliente y el servidor en forma de representaciones, como JSON o XML. Por ejemplo, cuando creamos un nuevo producto, podemos enviar los detalles del producto en formato JSON al servidor.

Principios REST

- **Sin estado:** REST es sin estado, lo que significa que cada solicitud contiene toda la información necesaria para procesarla. No se mantiene un estado de sesión en el servidor. Por lo tanto, cada solicitud es independiente y autónoma.
- **Interfaz uniforme:** REST promueve una interfaz uniforme entre el cliente y el servidor, lo que facilita la comprensión y el uso de la API. Esto significa que los clientes pueden interactuar con diferentes servicios de la misma manera, lo que simplifica el desarrollo y la integración.

Vistas en Django

Al pasar de SSR a APIREST, el uso de las vistas cambian ya que devolverán respuestas en formato JSON o XML.

DRF nos ofrecen tres tipos de vistas:

Tipo de Vista	Características
FBV (Function-Based Views)	Usa funciones para manejar solicitudes. Simples, pero requieren más código.
CBV (Class-Based Views)	Usa clases para definir lógica. Más estructuradas y reutilizables.
ViewSets	Más avanzadas, permiten manejar automáticamente varias operaciones (list, create, retrieve, etc.).

Vistas en Django

1) Function-Based Views (FBV): Son vistas basadas en funciones y usan decoradores como `@api_view`.

```
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework import status
from .models import Producto
from .serializers import ProductoSerializer

@api_view(["GET", "POST"])
def producto_lista(request):
    if request.method == "GET":
        productos = Producto.objects.all()
        serializer = ProductoSerializer(productos, many=True)
        return Response(serializer.data)

    elif request.method == "POST":
        serializer = ProductoSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

- `@api_view(["GET", "POST"])` → Permite que la vista acepte solo GET y POST.
- `serializer.is_valid()` → Valida los datos antes de guardarlos.
- `return Response(serializer.data)` → Devuelve una respuesta JSON.

Vistas en Django

2) Class-Based Views (FBV): Las vistas basadas en clases (CBV) usan APIView y métodos como get(), post(), put(), delete().

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from .models import Producto
from .serializers import ProductoSerializer

class ProductoLista(APIView):
    def get(self, request):
        productos = Producto.objects.all()
        serializer = ProductoSerializer(productos, many=True)
        return Response(serializer.data)

    def post(self, request):
        serializer = ProductoSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
        return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

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

urlpatterns = [
    path("productos/", ProductoLista.as_view(),
         name="producto-lista"),
```

Vistas en Django

3) ViewSets (Más automáticos): Los ViewSets simplifican aún más las vistas RESTful al agrupar métodos (list, create, retrieve, update, destroy).

```
from rest_framework.viewsets import ModelViewSet
from .models import Producto
from .serializers import ProductoSerializer

class ProductoViewSet(ModelViewSet):
    queryset = Producto.objects.all()
    serializer_class = ProductoSerializer
```

- Maneja automáticamente GET, POST, PUT, DELETE, etc.
- Requiere un Router en urls.py..

```
from rest_framework.routers import DefaultRouter
from .views import ProductoViewSet

router = DefaultRouter()
router.register(r"productos", ProductoViewSet)

urlpatterns = router.urls
```

- Ahora la API tendrá todas las rutas CRUD automáticamente.

Serializadores en Django

La mayor diferencia respecto a SSR es la existencia de serializadores. Estos se usan para convertir objetos Python/Django (como modelos de base de datos) en JSON y viceversa. Es decir, sus funciones son:

- Convertir datos de modelos a JSON (serialización).
- Convertir datos JSON a objetos Django (deserialización).
- Validar datos antes de guardarlos en la base de datos.

Serializadores en Django

Hay dos tipos de serializadores en DRF:

- **Serializador de modelo:** `serializers.ModelSerializer`

- Se basa en un **modelo Django**, generando los campos automáticamente.
- Es el más usado porque reduce código.

```
from rest_framework import serializers
from .models import Producto

class
ProductoSerializer(serializers.ModelSerializer):
    class Meta:
        model = Producto
        fields = '__all__'
```

- **Serializador Manual:** `serializers.Serializer`

- Se define cada campo manualmente.
- Se usa cuando no hay un modelo Django asociado.

```
from rest_framework import serializers

class ProductoSerializer(serializers.Serializer):
    id = serializers.IntegerField(read_only=True)
    nombre = serializers.CharField(max_length=100)
    precio = serializers.DecimalField(max_digits=10, decimal_places=2)
```

Validaciones en Django

- **Validación en el modelo (models.py):**

- Para restricciones universales (por ejemplo, valores mínimos/máximos).
- Para validaciones automáticas sin importar desde dónde venga la solicitud (admin, API, consola).
- Cuando necesitas asegurarte de que los datos en la base de datos sean siempre correctos.
- Si la validación depende de otros datos no disponibles en el modelo.
- Si necesitas enviar mensajes de error personalizados al usuario final.

```
from django.core.validators import MinValueValidator, MaxValueValidator
from django.db import models

class Producto(models.Model):
    nombre = models.CharField(max_length=100, unique=True) # Evita nombres repetidos
    precio = models.DecimalField(
        max_digits=10,
        decimal_places=2,
        validators=[MinValueValidator(0)]) # No puede ser negativo
    stock = models.IntegerField(
        validators=[MinValueValidator(0), MaxValueValidator(1000)]) # Entre 0 y 1000
```

Validaciones en Django

- **Validación en el Serializador (serializers.py):**

- Cuando necesitas validaciones específicas para la **API REST**.
- Si quieres mensajes de error personalizados para el usuario.
- Cuando la validación depende de múltiples campos.
- Si la validación debería aplicarse a **toda la aplicación**, no solo a la API.

```
from rest_framework import serializers
from .models import Producto

class ProductoSerializer(serializers.ModelSerializer):
    class Meta:
        model = Producto
        fields = '__all__'

    # Validación en un campo específico
    def validate_precio(self, value):
        if value < 1:
            raise serializers.ValidationError("El precio debe ser mayor a 1.")
        return value

    # Validación a nivel de objeto (combinación de campos)
    def validate(self, data):
        if data['precio'] > 10000 and data['stock'] < 5:
            raise serializers.ValidationError("Si el precio > 10,000, el stock debe ser al menos 5.")
        return data
```

Validaciones en Django

- **Validación en la vista (view.py):**

- Cuando necesitas validar datos según el usuario autenticado.
- Si la validación no tiene sentido en el nivel del modelo o del serializador.
- Si la validación es algo **básico (restricciones de base de datos)**, mejor hacerla en el modelo o serializador.

```
from rest_framework import viewsets, status
from rest_framework.response import Response
from .models import Producto
from .serializers import ProductoSerializer

class ProductoViewSet(viewsets.ModelViewSet):
    queryset = Producto.objects.all()
    serializer_class = ProductoSerializer

    def create(self, request, *args, **kwargs):
        data = request.data

        # Verificar si ya existe un producto con el mismo nombre
        if Producto.objects.filter(nombre=data.get("nombre")).exists():
            return Response({"error": "Ya existe un producto con este nombre."},
                           status=status.HTTP_400_BAD_REQUEST)

        # Si todo está bien, proceder con la creación
        return super().create(request, *args, **kwargs)
```

Validaciones en Django

Escenario	Modelo (models.py)	Serializador (serializers.py)	Vista (views.py)
Validar valores mínimos o máximos	✓	✓	✗
Asegurar restricciones de base de datos (por ejemplo, unique=True)	✓	✗	✗
Validar combinaciones de campos (ejemplo: precio > 10000 → stock >= 5)	✗	✓	✗
Personalizar mensajes de error	✗	✓	✓
Validar datos según información en la base de datos	✗	✗	✓
Validar datos en función del usuario (ejemplo: verificar permisos)	✗	✗	✓
Evitar duplicados en la base de datos	✗	✗	✓
Validar datos antes de guardarlos en la API	✗	✓	✓

OpenApi

OpenAPI es una especificación para describir y documentar APIs de servicios web. Proporciona un conjunto de reglas y formatos para describir las operaciones disponibles, los parámetros de entrada, los esquemas de datos de entrada y salida, y otros aspectos de una API web. OpenAPI permite a los desarrolladores comprender rápidamente cómo interactuar con una API y facilita la generación de documentación automática, así como la generación de clientes y servidores en diferentes lenguajes de programación.



OpenApi

- **Descripción de la API:** OpenAPI permite describir los endpoints de la API, los parámetros que aceptan, los métodos HTTP permitidos y la estructura de los datos de solicitud y respuesta. Esto se hace utilizando un documento en formato JSON o YAML que sigue la especificación de OpenAPI.
- **Estándar y Legibilidad:** Al utilizar OpenAPI, las APIs se describen de una manera coherente y fácil de entender. Esto facilita que los desarrolladores comprendan cómo interactuar con la API sin necesidad de leer extensa documentación.
- **Generación de Documentación Automática:** Con la especificación de OpenAPI, es posible generar automáticamente documentación interactiva para la API, incluyendo una interfaz de usuario que permite probar los endpoints directamente desde el navegador.
- **Validación de la API:** Las herramientas compatibles con OpenAPI pueden realizar validaciones automáticas del cumplimiento de la especificación, lo que ayuda a detectar posibles errores en la implementación de la API.
- **Compatibilidad con Herramientas y Frameworks:** OpenAPI es ampliamente compatible con diversas herramientas y frameworks, lo que permite su integración con diferentes tecnologías y entornos de desarrollo.



OpenApi

```
openapi: 3.0.3
info:
  title: API Shops
  version: 1.0.0
  description: API de productos de una tienda virtual
paths:
  /shops/categories/:
    get:
      operationId: shops_categories_list
      parameters:
        - name: page
          required: false
          in: query
          description: A page number within the paginated result set.
          schema:
            type: integer
      tags:
        - shops
      security:
        - cookieAuth: []
        - basicAuth: []
        - {}
    responses:
      '200':
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/PaginatedCategoryListCreateList'
            description: ''
```

Versión del estándar.
La 3 es la última.

Descripción general del API.

Se define para cada servicio que la API exponga, indicando métodos que atiende, parámetros, tags y código de estados de respuestas.

Swagger

OpenAPI es una especificación para describir y documentar APIs de servicios web. Proporciona un conjunto de reglas y formatos para describir las operaciones disponibles, los parámetros de entrada, los esquemas de datos de entrada y salida, y otros aspectos de una API web. OpenAPI permite a los desarrolladores comprender rápidamente cómo interactuar con una API y facilita la generación de documentación automática, así como la generación de clientes y servidores en diferentes lenguajes de programación.



Swagger

Originalmente, Swagger era el nombre de un proyecto de código abierto creado por SmartBear Software, pero posteriormente se renombró como "OpenAPI Specification". **Por eso es habitual que la gente use Swagger y OpenApi como sinónimos, pero no lo son.** OpenApi es la especificación y Swagger herramientas que facilitan documentar la API usando OpenApi.

Swagger es un conjunto de herramientas para el diseño y documentación de APIs RESTful. Estas son algunas de ellas:

- **Swagger UI:** Es una interfaz de usuario generada automáticamente que permite visualizar y probar APIs RESTful de forma interactiva. Proporciona una documentación dinámica que describe los endpoints, los métodos HTTP permitidos, los parámetros de entrada, las respuestas esperadas y otros detalles de la API.
- **Swagger Editor:** Es una herramienta en línea que permite escribir, validar y previsualizar especificaciones de API en formato Swagger (ahora OpenAPI Specification). Proporciona un editor de texto enriquecido con resaltado de sintaxis y validación en tiempo real para ayudar a crear y editar la especificación de la API.
- **Swagger Codegen:** Es una herramienta que permite generar automáticamente clientes de API en diferentes lenguajes de programación a partir de una especificación de API en formato Swagger. Esto facilita la creación de clientes de API consistentes y compatibles con la especificación de la API.



COMILLAS
UNIVERSIDAD PONTIFICIA

ICAI ICADE CIHS

Swagger-IU

API Shops 1.0.0 OAS 3.0

/api/schema/

API de productos de una tienda virtual

[Authorize](#)

shops

GET	/shops/categories/	🔒
POST	/shops/categories/	🔒
GET	/shops/categories/{id}/	🔒
PUT	/shops/categories/{id}/	🔒
PATCH	/shops/categories/{id}/	🔒
DELETE	/shops/categories/{id}/	🔒

OpenApi en Django

drf-spectacular es una [biblioteca](#) que genera automáticamente documentación OpenAPI (Swagger) para tu API REST en Django REST Framework (DRF).

1) Instalar: *pip install drf-spectacular*

2) En el fichero “settings.py”:

- Añade en INSTALLED_APPS, 'drf_spectacular'.
- Añade en REST_FRAMEWORK = {
 'DEFAULT_SCHEMA_CLASS': 'drf_spectacular.openapi.AutoSchema',
}

3) Añade rutas para la documentación:

```
from drf_spectacular.views import SpectacularAPIView, SpectacularSwaggerView, SpectacularRedocView
```

OpenApi en Django

4) Añadir decoradores a las views.py

```
@extend_schema_view(  
    list=extend_schema(  
        summary="Obtener lista de productos",  
        description="Este endpoint devuelve una lista de todos los productos almacenados en la base de datos.",  
        parameters=[  
            OpenApiParameter(name="nombre", type=str, description="Filtrar productos por nombre"),  
            OpenApiParameter(name="precio_min", type=float, description="Filtrar productos por precio mínimo"),  
            OpenApiParameter(name="precio_max", type=float, description="Filtrar productos por precio máximo"),  
        ],  
        responses={200: ProductoSerializer(many=True)} # Indica que devuelve una lista de productos  
    )  
)  
  
class ProductoViewSet(viewsets.ModelViewSet):
```

OpenApi en Django

@extend_schema_view(list=extend_schema(...))

Se usa para anotar el método list dentro del ViewSet.

Define un resumen, descripción y los parámetros aceptados.

parameters=[OpenApiParameter(...)]

Especifica los parámetros opcionales de consulta (query params).

En este caso, nombre, precio_min y precio_max.

responses={200: ProductoSerializer(many=True)}

Indica que la respuesta HTTP 200 devolverá una lista de productos en JSON.