ChatGPT

Tutorial Introductorio: Desarrollo de una Plataforma de Compras Online (API REST, Django, JWT, OpenAPI, Microservicios)

Introducción: Este tutorial está dirigido a un estudiante de Ingeniería en Sistemas con conocimientos mínimos de desarrollo de software, que junto a su grupo debe implementar una plataforma de compras online. A lo largo del documento se explicarán de manera clara (con ejemplos prácticos orientados a Visual Studio Code en Windows) los conceptos clave necesarios: desde **API** y **API REST**, pasando por la creación de una **API REST con Django y Django REST Framework**, la implementación de **JWT para autenticación**, la **documentación de la API con OpenAPI/Swagger**, hasta las **buenas prácticas de arquitectura** (microservicios) para un portal de compras con gestión de stock y logística. Al final se listan recursos útiles y videos recomendados para cada tema. El contenido está estructurado con un índice temático para facilitar su consulta.

Índice de Contenidos

1. ¿Qué es una API?

Definición: *API* son las siglas en inglés de *Application Programming Interface*, que en español significa Interfaz de Programación de Aplicaciones. En términos simples, una API es un **conjunto de funciones, métodos o reglas** que permiten que dos aplicaciones diferentes se comuniquen entre sí para compartir datos o funcionalidades [1]. En otras palabras, es como un puente que conecta sistemas distintos. Por ejemplo, cuando usas una aplicación móvil de clima y esta obtiene datos desde un servidor remoto, esa comunicación se realiza a través de una API: la app hace peticiones (solicita cierta información) y el servidor responde con los datos en un formato entendible (por lo general JSON).

APIs web y métodos HTTP: En el contexto de aplicaciones web, las APIs suelen exponerse mediante **endpoints HTTP** (URLs) y se interactúa con ellas usando los métodos del protocolo HTTP (a veces llamados *verbos HTTP*). Los cuatro métodos fundamentales son:

- **GET:** obtener o consultar un recurso. Por convención, las peticiones GET **sólo deben leer datos** (sin efectos secundarios en el servidor). Por ejemplo, `GET /productos` puede retornar la lista de productos. Según MDN, "el método GET solicita una representación de un recurso específico".
- **POST:** crear un recurso nuevo o realizar una acción que modifica el estado en el servidor. Por ejemplo, `POST /productos` podría crear un nuevo producto (enviando los datos en el cuerpo

de la petición). Este método *no* es idempotente (múltiples llamadas pueden crear duplicados). En palabras simples, POST suele usarse para **alta de datos**. MDN indica que el método POST "se utiliza para enviar una entidad a un recurso específico, causando a menudo un cambio de estado o efectos secundarios en el servidor".

- **PUT:** actualizar por completo un recurso existente, enviando la nueva representación. Por ejemplo, `PUT /productos/123` reemplazaría todos los datos del producto 123 con los proporcionados en la petición. Las operaciones PUT **son idempotentes**, es decir, si se realiza la misma petición varias veces, el resultado final en el servidor es el mismo. Según MDN, PUT "reemplaza todas las representaciones actuales del recurso de destino con la carga útil de la petición". (Nota: Existe también **PATCH** para actualizaciones parciales, pero para simplificar nos enfocamos en PUT para actualizaciones generales).
- **DELETE:** borrar un recurso. Por ejemplo `DELETE /productos/123` eliminaría el producto con ID 123. Como su nombre sugiere, este método **elimina** el recurso indicado y también es idempotente (borrar dos veces no produce error, la primera elimina y la segunda ya no encuentra el recurso). MDN resume que DELETE "borra un recurso en específico".

Cada método HTTP tiene una semántica específica. En el desarrollo de APIs REST (se verá en la siguiente sección) es importante adherirse a estas convenciones para que la API sea intuitiva. Además, las respuestas a estas peticiones suelen incluir *códigos de estado HTTP* estándar para indicar el resultado (por ejemplo, 200 OK, 201 Created, 404 Not Found, 400 Bad Request, etc.).

Ejemplo práctico: Supongamos que estamos construyendo la plataforma de compras online. Un **endpoint** posible de la API podría ser `/api/productos`.
- Un `GET /api/productos` devolvería la lista de productos disponibles (por ejemplo en formato JSON).
- Un `POST /api/productos` permitiría agregar un nuevo producto (enviando los detalles en JSON en el cuerpo de la petición).
- Un `GET /api/productos/123` obtendría la información del producto con ID 123.
- Un `PUT /api/productos/123` actualizaría completamente ese producto (por ejemplo cambiando su precio, descripción, etc.).
- Un `DELETE /api/productos/123` eliminaría el producto 123 del catálogo.

Como se ve, las APIs suelen trabajar con **URLs que representan recursos** (en plural) y métodos que definen la acción sobre esos recursos. Esta consistencia es clave para el diseño de APIs limpias.

2. ¿Qué es una API REST y cómo funciona?

El término **API REST** se refiere a un estilo particular de arquitectura para APIs web. *REST* significa *Representational State Transfer* (Transferencia de Estado Representacional). No es un protocolo ni un estándar formal, sino un conjunto de **principios de diseño** para servicios web. Una API que sigue estos principios se denomina **RESTful**.

Principios de REST: Las API REST se basan en recursos identificables mediante URLs y en el uso de los métodos HTTP estándar (como GET, POST, PUT, DELETE) para operar sobre esos recursos. Algunas características clave de una arquitectura RESTful son:

- **Arquitectura Cliente-Servidor:** Separación de preocupaciones entre el cliente (que consume la API, por ejemplo una aplicación front-end) y el servidor (que provee los recursos). El cliente y el servidor pueden evolucionar independientemente mientras la interfaz (API) se mantenga estable.

- **Comunicación Sin Estado:** Cada petición HTTP del cliente **debe contener toda la información necesaria** (por ejemplo, autenticación, datos, etc.) para que el servidor la entienda y la procese. El servidor no conserva estado de sesiones entre peticiones. Esto significa que, por ejemplo, si el cliente hace dos peticiones consecutivas, la segunda no depende de datos almacenados en el servidor de la primera petición (el servidor no "recuerda" cliente). Esto mejora la escalabilidad, ya que cualquier servidor puede atender cualquier petición independientemente.
- **Capacidad de Caché:** Las respuestas de la API deben indicar si son cacheables o no (mediante cabeceras HTTP). Si una respuesta es **cacheable**, el cliente (o intermediarios) pueden guardar los datos y reutilizarlos en peticiones posteriores, mejorando el rendimiento. REST promueve el uso eficiente de caché cuando aplica.
- **Interfaz Uniforme:** Es quizás el principio central de REST. Implica que todos los recursos se acceden de manera consistente, con una sintaxis homogénea. Algunos aspectos de la interfaz uniforme son: recursos identificados por URIs, manipulación de recursos a través de representaciones (por ejemplo, enviar JSON para modificar un recurso), mensajes autodescriptivos (cada petición y respuesta tiene suficiente información para describar cómo procesarla, mediante cabeceras, códigos de estado, etc.), y uso de hipermedia cuando sea apropiado (links en las respuestas que indican acciones siguientes posibles). Este último concepto se conoce como HATEOAS (Hypermedia As The Engine Of Application State), donde la respuesta de la API incluye enlaces para navegar a recursos relacionados.
- **Sistema en Capas:** La arquitectura puede componerse de capas (por ejemplo, servidores intermedios, balanceadores, proxies, etc.) de forma transparente para el cliente. Cada capa sólo se comunica con la adyacente. Esto permite añadir escalabilidad, seguridad (por ej. gateways), etc., sin que el cliente lo note.
- **Código bajo demanda (opcional):** El servidor podría proporcionar código ejecutable al cliente bajo demanda (por ejemplo, scripts JavaScript). Este principio es opcional y no es usado en todas las APIs REST.

En resumen, una **API RESTful** es una API web que cumple con estas condiciones. La ventaja de REST es que aprovecha las capacidades ya existentes de HTTP (métodos, códigos, caché) de forma sencilla y uniforme, lo que la hace ligera y adecuada para servicios modernos (en contraste con enfoques más complejos como SOAP). De hecho, REST se ha vuelto el estilo predominante para crear APIs web abiertas debido a su simplicidad y flexibilidad.

Caso práctico (REST en la plataforma de compras): Imaginemos los recursos principales de nuestra plataforma: *usuarios*, *productos*, *pedidos*, *envíos*, etc. Una API REST podría definir endpoints como:

- `/api/productos` – para crear (**POST**), listar (**GET**) productos, etc.
- `/api/productos/{id}` – para operaciones sobre un producto específico (consultar con GET, actualizar con PUT/PATCH, borrar con DELETE).
- `/api/usuarios/{id}/pedidos` – para listar los pedidos de un cierto usuario, o crear un nuevo pedido para ese usuario, etc.
- `/api/pedidos/{id}/seguimiento` – para consultar el estado de envío de un pedido determinado.

Siguiendo REST, cada URL representa claramente un **recurso**, y los métodos indican la acción. Las respuestas suelen manejarse en JSON. Por ejemplo, una respuesta a `GET /api/productos/123` podría ser un JSON como:

```
{
  "id": 123,
  "nombre": "Laptop XYZ",
```

```
  "precio": 750.00,
  "stock": 5,
  "descripcion": "Laptop XYZ de 15 pulgadas...",
  "categoria": "/api/categorias/5",
  "url": "/api/productos/123"
}
```

Obsérvese que podrían incluirse enlaces (por ejemplo, la categoría del producto como URL de recurso) siguiendo HATEOAS. Un desarrollador que use esta API, sin conocer detalles internos, puede seguir las URLs incluidas en las respuestas para navegar por los recursos disponibles.

En conclusión, REST es un modelo que aporta **estandarización**: si has usado una API REST, te resultará familiar usar otra distinta porque siguen patrones semejantes.

3. Creación de una API REST con Django y Django REST Framework

Ahora que entendemos qué es una API REST, veamos cómo **implementarla** usando Python con el framework Django y su extensión **Django REST Framework (DRF)**. Django es un framework web robusto de alto nivel en Python, y Django REST Framework es una biblioteca especializada que se integra con Django para facilitar la construcción de APIs REST de manera rápida y fácil.

Preparando el entorno (Windows + VS Code): Antes de codificar, debemos preparar un entorno de trabajo aislado (entorno virtual) e instalar las dependencias. A continuación, los pasos básicos:

1. **Crear y activar un entorno virtual:** Abre una terminal en VS Code (PowerShell o CMD en Windows) y ejecuta:

   ```
   > python -m venv env
   > env\Scripts\activate
   ```

   Esto crea un directorio `env` con el entorno virtual de Python y luego lo activa (notarás que el prompt cambia para mostrar `(env)` al inicio). En PowerShell puede que necesites usar `env\Scripts\Activate.ps1`.

2. **Instalar Django y Django REST Framework:** Con el entorno activado, instala las librerías necesarias usando pip:

   ```
   (env) > pip install django djangorestframework
   ```

   Esto descargará e instalará Django y DRF en tu entorno virtual.

3. **Iniciar un nuevo proyecto Django:** Ejecuta el comando:

   ```
   (env) > django-admin startproject tiendaOnline .
   ```

Este comando crea la estructura básica de un proyecto Django llamado "tiendaOnline". (El punto `.` al final indica que lo cree en el directorio actual; puedes omitirlo para que cree un subdirectorio). Tras este comando, deberías tener un árbol de archivos similar a:

```
tiendaOnline/  (directorio del proyecto)
    manage.py
    tiendaOnline/
        __init__.py
        settings.py
        urls.py
        asgi.py
        wsgi.py
```

1. **Crear una aplicación Django para la API:** Dentro del proyecto, las funcionalidades suelen agruparse en *apps*. Crea una nueva app (por ejemplo, llamada "tienda" para manejar nuestros recursos de la tienda):

   ```
   (env) > python manage.py startapp tienda
   ```

   Esto generará un directorio `tienda/` con archivos básicos (`models.py`, `views.py`, etc.). No olvides agregar esta app en el `settings.py` del proyecto, en la lista `INSTALLED_APPS`, junto con `'rest_framework'` para habilitar DRF:

```
# tiendaOnline/settings.py
INSTALLED_APPS = [
    ...,
    'rest_framework',   # habilitar Django REST Framework
    'tienda',           # nuestra app "tienda"
]
```

1. **Definir el modelo de datos (models.py):** Para ejemplificar, crearemos un modelo de Producto simple en `tienda/models.py`:

```
# tienda/models.py
from django.db import models

class Producto(models.Model):
    nombre = models.CharField(max_length=100)
    descripcion = models.TextField(blank=True)
    precio = models.DecimalField(max_digits=10, decimal_places=2)
    stock = models.IntegerField(default=0)

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

Este modelo define la estructura de la tabla de productos en la base de datos (campos para nombre, descripción, precio y stock). Django por defecto usará SQLite como base de datos, que es suficiente para desarrollo.

Ahora hay que crear las migraciones y aplicar estos cambios a la base de datos:

```
(env) > python manage.py makemigrations
(env) > python manage.py migrate
```

Esto genera y ejecuta las migraciones, creando la tabla `tienda_producto` en la base de datos.

1. **Crear un Serializador (serializers.py):** DRF utiliza serializadores para convertir instancias de modelos Django a JSON y viceversa. Creamos `tienda/serializers.py`:

```
# tienda/serializers.py
from rest_framework import serializers
from .models import Producto

class ProductoSerializer(serializers.ModelSerializer):
    class Meta:
        model = Producto
        fields = '__all__'  # serializar todos los campos del modelo
        # fields = ['id', 'nombre', 'descripcion', 'precio', 'stock']  #
opción explícita
```

Con `ModelSerializer` DRF facilita crear un serializador basado en el modelo `Producto`. `__all__` indicamos que incluya todos los campos del modelo. (También podríamos listar campos específicos).

1. **Crear las Vistas (views.py):** Las *vistas* manejan las peticiones y responden usando los serializadores. DRF provee vistas genéricas y viewsets para ahorrar código. En nuestro caso, usaremos un *ViewSet* para el modelo Producto, que automáticamente manejará las operaciones CRUD básicas:

```
# tienda/views.py
from rest_framework import viewsets
from .models import Producto
from .serializers import ProductoSerializer

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

Con apenas estas líneas, hemos creado una vista que soporta GET (listado y detalle), POST (crear), PUT/PATCH (actualizar) y DELETE (eliminar) para el recurso Producto. `ModelViewSet` de DRF infiere las acciones a partir del queryset y el serializer provistos.

1. **Configurar las URLs de la API:** Django centraliza las rutas en el archivo `urls.py` del proyecto. Debemos incluir las rutas para nuestro viewset. Podemos usar un enrutador automático de DRF para generar las rutas estándar de una API REST. Edita `tiendaOnline/urls.py`:

```
# tiendaOnline/urls.py
from django.contrib import admin
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from tienda.views import ProductoViewSet

router = DefaultRouter()
router.register(r'productos', ProductoViewSet, basename='producto')

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

Aquí registramos `ProductoViewSet` bajo la ruta `productos`. El router generará endpoints como: - `/api/productos/` – lista de productos (GET) y creación (POST). - `/api/productos/{id}/` – detalle de producto (GET), actualización (PUT/PATCH) o eliminación (DELETE).

Django REST Framework incluso añade una ruta de navegación por API por defecto en la raíz del router, así que `/api/` nos podría mostrar un índice de endpoints disponibles.

1. **Probar la API en desarrollo:** Arranca el servidor de desarrollo de Django:

   ```
   (env) > python manage.py runserver
   ```

   Por defecto en `http://127.0.0.1:8000/`. Navega a `http://127.0.0.1:8000/api/productos/`. Si todo está bien, deberías ver (gracias a DRF) una interfaz de navegador para la API. Inicialmente la lista estará vacía (`[]`) porque no hay productos creados. Puedes verificar también yendo a la ruta del admin de Django (`/admin/`) – pero primero crea un súperusuario con `python manage.py createsuperuser` – e ingresar un par de productos mediante la interfaz de administrador para tener datos de prueba.

La **API Browsable** de DRF: Una gran característica es que en modo debug, DRF provee automáticamente una interfaz web navegable para la API. Desde el navegador puedes hacer POST, PUT, DELETE llenando formularios, lo cual es útil para pruebas rápidas sin necesidad de herramientas externas.

1. **Pruebas de la API (via HTTP):** También podemos probar usando herramientas como curl, Postman o similares. Por ejemplo, para consultar la lista de productos via curl:

   ```
   > curl -X GET http://127.0.0.1:8000/api/productos/
   ```

   Debería responder con JSON, por ejemplo:

```
[]
```

(una lista vacía). Si agregaste un producto de prueba, verás ese objeto en JSON. Para probar un *POST* (crear) vía curl:

```
> curl -X POST http://127.0.0.1:8000/api/productos/ \
     -H "Content-Type: application/json" \
     -d "{\"nombre\": \"Producto1\", \"descripcion\": \"demo\",
\"precio\": 100.0, \"stock\": 10}"
```

Esto enviaría un JSON con los campos necesarios. La respuesta debería incluir los datos creados (y un código 201 Created). Alternativamente, con HTTPie (una herramienta de línea de comando más amigable que curl) sería:

```
> http POST :8000/api/productos/ nombre="Producto1" descripcion="demo"
precio:=100.0 stock:=10
```

(HTTPie permite esta sintaxis conveniente).

¡Y listo! Hemos creado una API REST básica. Con muy pocas líneas, DRF generó automáticamente todos los endpoints CRUD para *productos* siguiendo las convenciones REST. En un entorno real, podríamos añadir validaciones adicionales, permisos, autenticación, etc., pero esos temas se verán más adelante (JWT en la siguiente sección).

¿Qué logramos? Un cliente puede ahora interactuar con nuestro servidor de Django a través de esta API. Por ejemplo, una aplicación frontend (JavaScript) podría pedir la lista de productos con una simple petición AJAX a `/api/productos/`, obtener un JSON y renderizarlo en la interfaz de usuario. Todo esto sin generar HTML en Django, sino usando la API como capa intermedia de datos.

4. ¿Qué es JWT (JSON Web Token) y cómo implementarlo en Django para autenticación?

Cuando construimos una API, tarde o temprano necesitaremos **autenticar usuarios** y proteger ciertos endpoints (por ejemplo, que solo usuarios registrados puedan crear pedidos, o que solo administradores puedan agregar productos). Una forma muy popular de manejar la autenticación en APIs modernas es usando **JWT (JSON Web Tokens)**.

Concepto de JWT: Un JWT es esencialmente un **token** (una cadena de texto) que representa la autenticación de un usuario de forma **autocontenida**. "Autocontenida" significa que el token lleva en sí mismo la información del usuario (o un identificador) y una firma digital para verificar que no ha sido alterado. Un JWT está compuesto por **tres partes**: encabezado, carga útil y firma, concatenadas y codificadas en Base64 URL-safe. Por ejemplo, un token puede verse así:

```
xxxxx.yyyyy.zzzzz
```

Donde: - `xxxxx` es el **Header** (encabezado) codificado, que usualmente solo indica el tipo de token (JWT) y el algoritmo de firma utilizado (por ejemplo HS256, RS256).
- `yyyyy` es el **Payload** (carga útil) codificado, que contiene los *claims* o declaraciones: datos como el ID de usuario, nombre, roles, y fechas de expiración, etc. Por ejemplo, un claim estándar es `exp` (expiry) que indica hasta cuándo es válido el token.
- `zzzzz` es la **Firma** digital, obtenida aplicando un algoritmo criptográfico al header y payload, usando una clave secreta (en HS256 una clave secreta compartida; en RS256 la clave privada del servidor). La firma permite al servidor verificar que el token fue emitido por él mismo y que no ha sido modificado en el camino.

El servidor emite el token JWT típicamente cuando el usuario inicia sesión con usuario/contraseña correctamente. Luego, el cliente almacena ese token (por ejemplo en memoria o almacenamiento local seguro) y lo envía en **cada petición subsiguiente** a la API para demostrar su identidad. Por convención, el token se envía en una cabecera HTTP:

```
Authorization: Bearer <token_jwt>
```

Así, el servidor, en cada petición recibida, puede verificar la firma del token (usando el secreto o clave pública correspondiente) y, si es válido, extraer la identidad del usuario desde el payload del JWT. Esto se hace **sin consultar una base de datos ni mantener sesiones en servidor**, lo que hace a JWT una solución **stateless** y muy adecuada para arquitecturas distribuidas y microservicios.

Flujo típico de autenticación con JWT: ① El usuario envía sus credenciales (ej. email y contraseña) al servidor (por ejemplo a un endpoint `/api/login/`). ② El servidor verifica esas credenciales; si son válidas, genera un JWT firmado que incluye la identidad del usuario (por ejemplo, su ID o nombre de usuario) y lo devuelve en la respuesta. ③ El cliente (p. ej. la aplicación frontend) guarda ese token. ④ En adelante, para cada petición a endpoints protegidos, el cliente envía el token en la cabecera `Authorization`. ⑤ El servidor recibe el token, lo verifica (firma y expiración) y, si es válido, considera al usuario autenticado y permite realizar la acción solicitada. Este proceso reemplaza al esquema tradicional de sesión (cookies) en muchos servicios de APIs.

Implementación en Django REST Framework: DRF no incluye JWT por defecto, pero existen paquetes de terceros que lo integran fácilmente. Uno muy usado es **djangorestframework-simplejwt**. Veamos cómo incorporarlo al proyecto:

1. **Instalar la librería JWT:**

   ```
   (env) > pip install djangorestframework-simplejwt
   ```

   (Asumimos que ya tienes DRF instalado).

2. **Configurar DRF para usar JWT:** En el `settings.py`, agregar SimpleJWT como autenticador por defecto:

   ```python
   # tiendaOnline/settings.py
   REST_FRAMEWORK = {
       'DEFAULT_AUTHENTICATION_CLASSES': (
   ```

```
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ),
    # (DEFAULT_PERMISSION_CLASSES se puede configurar también)
}
```

Esto le indica a DRF que espere tokens JWT en las peticiones para la autenticación.

3. **Crear las vistas para obtener el token:** SimpleJWT provee vistas ya hechas para obtener y refrescar tokens. Podemos incluirlas en `urls.py`:

```
from rest_framework_simplejwt.views import TokenObtainPairView,
TokenRefreshView

urlpatterns = [
    ...,
    path('api/token/', TokenObtainPairView.as_view(),
name='token_obtain_pair'),
    path('api/token/refresh/', TokenRefreshView.as_view(),
name='token_refresh'),
]
```

4. El endpoint `/api/token/` aceptará las credenciales de usuario (ej. nombre de usuario y password) via POST y, si son correctas, devolverá un JSON con `access` (el JWT de acceso) y `refresh` (un token de refresco). El token de acceso generalmente se configura para que expire en poco tiempo (ej: 5 minutos) por seguridad, mientras que el refresh token puede durar más (ej: 24 horas) y sirve para obtener un nuevo access token sin pedir credenciales de nuevo.
5. El endpoint `/api/token/refresh/` a su vez acepta un refresh token válido y retorna un nuevo access token.

SimpleJWT utiliza por defecto el modelo de usuarios de Django. Asegúrate de tener creado al menos un usuario (puedes reutilizar el súperusuario que creamos antes). Para probar, puedes usar herramientas como curl o Postman:

```
> curl -X POST http://127.0.0.1:8000/api/token/ -H "Content-Type:
application/json" -d "{\"username\": \"tu_usuario\", \"password\":
\"tu_password\"}"
```

Debería responder con algo como:

```
{
  "refresh": "eyJ0eXAiOiJK... (muy largo) ...",
  "access": "eyJ0eXAiOiJKV1QiLCJhbGciOi... (muy largo) ..."
}
```

Copia el valor de "access" (es el JWT). Ahora puedes hacer una petición autenticada, por ejemplo:

```
> curl -H "Authorization: Bearer <TU_TOKEN_ACCESS>" http://127.0.0.1:8000/
api/productos/
```

Si el token es válido, el servidor decodificará que eres el usuario autenticado correspondiente y devolverá la lista de productos. Si intentas acceder a un endpoint protegido sin token o con token inválido, obtendrás típicamente un error 401 Unauthorized.

1. **Proteger las vistas/recursos con autenticación:** Por defecto, nuestras viewsets pueden ser accesibles sin autenticar (DRF pone `AllowAny` permission globalmente). Para requerir login, podemos establecer permisos globalmente o por view. Por ejemplo, para exigir login en todas las APIs, en `settings.py` añadir:

   ```
   REST_FRAMEWORK = {
       'DEFAULT_AUTHENTICATION_CLASSES': (... JWTAuthentication ...),
       'DEFAULT_PERMISSION_CLASSES': (
           'rest_framework.permissions.IsAuthenticated',
       ),
   }
   ```

   Esto hará que *todas* las vistas de DRF requieran un usuario autenticado (token válido). Alternativamente, podemos decorar o configurar cada ViewSet individual con un `permission_classes = [IsAuthenticated]` o similar. Por simplicidad, supongamos que toda nuestra API requiere autenticación salvo algunos endpoints públicos.

Con esta configuración, hemos implementado JWT en nuestro backend. Ahora la seguridad es *stateless*: el servidor no guarda sesión del usuario, sólo valida cada token entrante. Si por ejemplo escalamos el backend a varios servidores, cada uno puede validar tokens de forma independiente con la misma clave secreta.

Notas de seguridad y uso de JWT: - Mantén la clave secreta (SECRET_KEY de Django o la específica para JWT si configuras una) bien segura. Si alguien la obtuviera, podría falsificar tokens. - Usar HTTPS es obligatorio al usar JWT, ya que si no, un atacante podría interceptar el token (que va por header) y usarlo (JWT no va ligado a dominio como las cookies, por lo que robar un JWT permite impersonar al usuario). - El token de **refresh** sirve para obtener nuevos tokens de acceso sin pedir credenciales constantemente. Asegúrate de proteger bien el refresh y posiblemente invalidarlo (rotarlo) tras su uso. SimpleJWT ya maneja expiración y rotación según configuración. - **Invalidación de tokens:** una desventaja de JWT vs sesiones tradicionales es que, una vez emitido el token, es difícil "revocarlo" (por ejemplo, si el usuario cierra sesión, el token JWT seguirá siendo válido hasta que expire). Hay estrategias para invalidar (listas negras de tokens, o reducir tiempos de expiración), pero añaden complejidad. En aplicaciones simples académicas esto no suele ser crítico. - Opcionalmente, podrías almacenar el JWT en el cliente en una cookie HttpOnly segura para emular el comportamiento de sesión (aunque si vas a usar cookies, podrías usar la autenticación de sesión de Django directamente; JWT es más útil cuando el cliente es totalmente independiente, ej. una SPA o app móvil).

JWT y nuestro proyecto de compras: Con JWT implementado, podemos restringir ciertas operaciones. Por ejemplo, *obtener la lista de productos* quizás sea público, pero *crear/editar productos* debería requerir un usuario vendedor autenticado. *Realizar una compra* requiere un usuario comprador autenticado, etc. JWT nos permite distribuir esta lógica de autorización a los microservicios también, como veremos: el mismo token JWT podría ser usado en el servicio de portal de compras, en el de stock y en el de

logística, de forma que cada microservicio valida el token y confía en la identidad del usuario sin necesidad de compartir sesión.

5. ¿Qué es OpenAPI y cómo documentar una API Django?

Desarrollar la API es solo parte del trabajo; también es crucial **documentarla** para que otros (o tu propio equipo) sepan cómo usarla. Aquí entra **OpenAPI**. OpenAPI Specification (OAS), antes conocida como *Swagger*, es un estándar para describir de forma **estructurada** las APIs REST. Básicamente, es un formato (generalmente un archivo YAML o JSON) que lista todos los endpoints de la API, los métodos, qué parámetros esperan, qué respuestas devuelven, esquemas de datos, códigos de error, etc., todo en un solo documento legible tanto por humanos como por máquinas.

Estructura de OpenAPI: Un archivo OpenAPI contiene típicamente: - **Información general:** título de la API, versión, descripción, contacto, etc. - **Listado de rutas o paths:** para cada endpoint (p. ej. `/api/productos`), se documentan los métodos soportados (GET, POST, ...), qué hace cada uno, qué parámetros o body esperan (por ejemplo, query parameters, parámetros de ruta, esquema JSON de entrada), y qué posibles respuestas devuelven (códigos 200, 400, 404, etc., con sus respectivos cuerpos o esquemas de datos). - **Esquemas o modelos (components/schemas):** definiciones de objetos reutilizables, por ejemplo, definir el esquema *Producto* con sus campos (id, nombre, precio...). Esto evita repetir la estructura en cada endpoint. - **Seguridad:** métodos de auth utilizados (por ejemplo JWT Bearer, API keys, OAuth) y cómo se aplican. - **Otros metadatos:** servidores (URLs base de la API), ejemplos de uso, etc.

En resumen, un archivo OpenAPI describe **las capacidades de la API** de forma estandarizada, sin necesidad de leer el código o tener documentación dispersa. Según una guía, permite comprender fácilmente qué hace cada endpoint, qué datos enviar y qué esperar como respuesta [2].

Un pequeño ejemplo (simplificado) en YAML para ilustrar una sección OpenAPI de nuestra API de productos podría ser:

```yaml
openapi: "3.0.3"
info:
  title: "API de Tienda Online"
  version: "1.0.0"
paths:
  /api/productos/:
    get:
      summary: "Lista todos los productos"
      responses:
        '200':
          description: "Éxito. Retorna la lista de productos."
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: "#/components/schemas/Producto"
    post:
      summary: "Crea un nuevo producto"
      requestBody:
```

```
        required: true
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Producto"
      responses:
        '201':
          description: "Producto creado exitosamente."
        '400':
          description: "Datos inválidos."
  /api/productos/{id}/:
    get:
      summary: "Obtiene un producto por ID"
      parameters:
        - name: id
          in: path
          required: true
          schema:
            type: integer
      responses:
        '200':
          description: "Éxito. Retorna el producto solicitado."
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Producto"
        '404':
          description: "No se encontró un producto con ese ID."
components:
  schemas:
    Producto:
      type: object
      properties:
        id:
          type: integer
        nombre:
          type: string
        descripcion:
          type: string
        precio:
          type: number
          format: float
        stock:
          type: integer
      required: ["nombre", "precio"]
```

No te asustes por la sintaxis – la idea principal es ver cómo se estructura: se definen las rutas (`/api/productos/`, `/api/productos/{id}/`), para cada ruta los métodos (get, post, etc.) con sus detalles. En este ejemplo, describimos que `GET /api/productos/` retorna un 200 con un JSON array de *Producto*, y que `POST /api/productos/` espera un cuerpo JSON según el esquema *Producto*.

También definimos el esquema *Producto* en components, indicando sus campos. Esto sirve como documentación: un desarrollador leyendo esto sabe qué campos enviar y qué recibirá.

Swagger UI / Herramientas de documentación: OpenAPI brilla porque permite usar herramientas automáticas. Por ejemplo, *Swagger UI* y *ReDoc* son aplicaciones web (UI) que leen el archivo OpenAPI y generan una **documentación interactiva** de la API. Probablemente hayas visto alguna vez documentaciones donde puedes probar endpoints directamente desde el navegador: eso suele ser Swagger UI. Puedes alojar Swagger UI junto a tu API para que los desarrolladores vean los endpoints y hagan pruebas. ReDoc genera una bonita página de documentación estática.

Documentar API Django automáticamente: Es tedioso escribir a mano todo el YAML. Afortunadamente, DRF puede generar automáticamente (al menos una base) del esquema OpenAPI inspeccionando las vistas, modelos y serializadores. Existen bibliotecas como **drf-yasg** o **drf-spectacular** que integran la generación de esquemas OpenAPI en Django. Por ejemplo, usando *drf-yasg*, puedes agregar algo así en `urls.py`:

```
from drf_yasg.views import get_schema_view
from drf_yasg import openapi

schema_view = get_schema_view(
    openapi.Info(
        title="API Tienda Online",
        default_version='v1',
        description="Documentación de la API de la tienda online",
    ),
    public=True,
)

urlpatterns = [
    ...,
    path('api/docs/', schema_view.with_ui('swagger', cache_timeout=0),
name='schema-swagger-ui'),
]
```

Con esto, ejecutando el servidor y visitando `/api/docs/`, aparecerá la interfaz Swagger UI con la documentación de la API generada. Puedes probar endpoints desde allí mismo. Alternativamente, *drf-spectacular* genera un archivo schema.yml que puedes servir con ReDoc.

Mantener la documentación: Aunque las herramientas automáticas ayudan, es importante anotar nuestras vistas o serializadores con descripciones para enriquecer la documentación. DRF permite usar docstrings o el atributo `help_text` en campos, etc., para que el generador de esquema incluya esos detalles (por ejemplo, descripciones de cada campo, ejemplos, etc.). Una documentación limpia y clara reduce muchísimo la fricción para quienes consumen tu API.

En resumen, **OpenAPI/Swagger** nos proporciona un "manual de uso" formal para la API [3]. En proyectos grupales (como el tuyo), es muy útil acordar desde temprano el contrato de la API (quizás escribiendo primero el esquema OpenAPI) para que todos los miembros sepan qué endpoints existirán y cómo interactuarán los servicios entre sí, incluso antes de implementarlos. Además, durante el desarrollo y entrega, contar con documentación actualizada es un plus enorme.

6. Buenas prácticas de arquitectura y conexión entre microservicios

El enunciado del proyecto indica que el sistema de compras online está compuesto por **tres subsistemas o microservicios principales**: **Portal de Compras**, **Stock de Bienes y Servicios**, y **Transporte, Logística y Seguimiento**. Cada uno de estos cumple un rol específico dentro de la plataforma:

- **Portal de Compras:** es la aplicación frontal que usan los clientes finales. Debe permitir a los usuarios registrarse, iniciar sesión, buscar productos, realizar compras y hacer seguimiento de sus pedidos (hasta la entrega). Es la cara visible del sistema, posiblemente un sitio web o app móvil que consume las APIs de los otros servicios.
- **Servicio de Stock de Bienes y Servicios:** se encarga de la gestión de productos por parte de los vendedores. Aquí los vendedores pueden cargar sus productos para que aparezcan en el portal, actualizar stocks, etc. Además, este servicio expone API hacia el Portal para consultar disponibilidad y reservar stock cuando se realiza una compra. Los usuarios que acceden a este subsistema podrían compartir la base de datos de usuarios con el portal (para no duplicar registros), pero con un rol diferente (rol vendedor). Es decir, un mismo individuo podría tener una cuenta que le permite comprar en el portal y también vender en el sistema de stock, pero con permisos distintos según contexto.
- **Servicio de Transporte, Logística y Seguimiento:** se encarga de todo lo relacionado con los envíos de los pedidos. Calcula costos de envío según reglas (tipo de transporte, distancia, peso, etc.), coordina la entrega con transportistas y mantiene informado al Portal del estado de cada envío. Por ejemplo, cuando el Portal necesita cotizar el costo de envío de un carrito de compra, consulta a este servicio; una vez concretada la compra, el Portal genera un pedido de envío aquí, y luego este servicio enviará actualizaciones (eventos o notificaciones) de seguimiento que el Portal mostrará al usuario final.

Interacción entre los microservicios: Siguiendo el enunciado, el flujo de una compra involucra a los tres servicios: Cuando un usuario compra algo en el **Portal**, el portal debe 1) solicitar al servicio de **Stock** que reserve el stock de los productos comprados (para no venderlos a otro mientras se procesa el envío), y 2) registrar un nuevo envío en el servicio de **Transporte**. Luego, el servicio de **Transporte** se encargará de efectivamente gestionar la entrega; este a su vez consultará detalles de los productos al servicio de **Stock** (ej. dimensiones o ubicaciones de almacén) para optimizar el envío. Una vez el paquete está en camino, el servicio de **Transporte** enviará actualizaciones de estado al **Portal** (por ejemplo “en tránsito”, “en reparto”, “entregado”) para que el cliente pueda ver el seguimiento, y notificará al Portal cuando el producto ha sido entregado exitosamente. Si por alguna razón un pedido no llega y expira cierto tiempo, el servicio de Transporte también debería notificar la necesidad de liberar la reserva de stock en el servicio de Stock.

Diagrama: Interacción entre el Portal de Compras, el servicio de Stock y el servicio de Transporte en la arquitectura de microservicios. Como se aprecia, el **Portal** actúa como orquestador de la operación de compra: coordina con los otros dos servicios las acciones necesarias. Cada microservicio tiene su propia base de datos y lógica, y se comunican entre sí mediante llamadas a sus APIs (posiblemente llamadas REST síncronas). Veamos algunas **buenas prácticas** para esta arquitectura:

- **Definir responsabilidades claras y límites de servicio:** Cada microservicio debe centrarse en un conjunto de funcionalidades claramente definidas (principio de *Single Responsibility* aplicado a servicios). En nuestro caso: Portal se encarga de experiencia de usuario y lógica de negocio de compras; Stock maneja la información de productos y disponibilidad; Logística maneja envíos.

Esto reduce acoplamiento. Cada servicio debería tener **independencia de despliegue**: poder ser actualizado o escalado sin afectar a los demás.

- **Comunicación a través de APIs bien definidas:** Los microservicios se conectan mediante APIs. Es fundamental diseñar **contratos de API estables** para la interacción entre servicios. Por ejemplo, el Portal debe saber cómo solicitar una reserva de stock: podríamos definir un endpoint en el servicio de Stock, e.g. `POST /api/reservas/` con el detalle del producto y cantidad a reservar. El stock respondería si pudo reservar o no. Similarmente, para crear un envío en el servicio de Transporte, podría existir `POST /api/envios/` donde el portal envía la dirección de entrega, items, etc., y recibe un ID de seguimiento. Estos contratos (endpoints, formatos JSON, etc.) deben documentarse (¡aquí entra OpenAPI, incluso para servicios internos!). Idealmente, usar **nombres y estructuras consistentes** en las APIs de todos los servicios hace la integración más sencilla.
- **Sincronía vs asincronía:** Llamadas REST son típicamente síncronas (el Portal espera la respuesta inmediata de Stock y Transporte). Esto está bien para solicitar cosas en tiempo real (reserva stock, calcular envío). Pero para notificaciones (como actualizaciones de tracking o liberar stock tras un tiempo) puede ser útil usar comunicación **asíncrona** mediante mensajes o eventos. Por ejemplo, el servicio de Transporte podría enviar un evento "Entrega completada" a un sistema de mensajería (como RabbitMQ, Kafka) al que el Portal esté suscrito, o bien llamar a un endpoint del Portal (`POST /api/notificaciones/entrega`) para informarle. La asincronía desacopla aún más los servicios (no necesitan esperar respuesta inmediata) y es útil para eventos que pueden manejarse en segundo plano. Sin embargo, añade complejidad (gestionar colas, reintentos, etc.). En una primera implementación, probablemente bastará con llamadas directas con cierto manejo de errores.
- **Manejando la consistencia de datos:** En una arquitectura distribuida, mantener los datos consistentes es un reto. Ejemplo: el Portal reserva stock reduciendo el inventario disponible; ¿qué pasa si luego el servicio de Transporte informa que la entrega no se concretó (cliente no estaba) y se canceló el pedido? Debemos **liberar ese stock reservado**. ¿Quién hace esto y cómo? Según el enunciado, el servicio de Transporte sería el encargado de liberar stock tras cierto tiempo de reserva no concretada. Esto implica que Transporte debe comunicarse con Stock (vía API) para ajustar el inventario. Son lo que se llaman **operaciones compensatorias** o sagas, para mantener consistencia eventual entre servicios. Una buena práctica es definir claramente estos flujos de compensación: e.g., "ante cancelación de pedido o expiración, transporte llama a stock para sumar X unidades al inventario".
- **Autenticación y autorización distribuida:** Compartir identidad entre servicios es mencionado en el enunciado (usuarios que actúan en portal y stock). Una práctica común es tener un servicio central de autenticación (o utilizar JWT compartido). Por ejemplo, el Portal podría autenticar al usuario y emitir un JWT (como vimos en sección 4). Ese JWT luego puede ser usado para llamar a los otros servicios (Stock, Transporte) directamente, y esos servicios, confiando en la autoridad emisora del JWT (el portal, o un Auth server), validan el token y obtienen la identidad/rol. Así, un vendedor con JWT válido podría llamar al servicio de Stock directamente para cargar sus productos. Otra opción es que las comunicaciones entre ciertos servicios ocurran con *credenciales de servicio* (no las del usuario final) cuando es back-end a back-end. Por ejemplo, el Portal al llamar a Transporte podría usar un token especial para servicios internos, o acordar una API key segura, etc. Lo importante es **no transmitir datos sensibles sin autenticación** y asegurarse que cada servicio solo permita las acciones autorizadas (un usuario final no debería poder invocar directamente la API interna de stock para reservar lo que quiera, a menos que venga del portal autenticado en un flujo válido).
- **API Gateway (Opcional):** En arquitecturas microservicios a veces se coloca un *API Gateway* que centraliza las llamadas externas. Por ejemplo, el cliente final (app front-end) en vez de llamar a 3 servicios distintos, solo llama al Gateway, y este redirige internamente a Portal/Stock/Transporte según el caso. Para tu proyecto académico tal vez no sea necesario, pero es bueno mencionarlo.

Un gateway puede unificar autenticación, prevenir exponer servicios internos directamente, y simplificar llamadas de front-end (todas a un mismo dominio). Sin gateway, el front-end podría llamar al Portal para algunas cosas y a Transporte para otras directamente si cada uno expone sus APIs públicamente.

- **Bases de datos separadas:** Cada microservicio debe idealmente tener su propia base de datos, para evitar acoplamiento a nivel de datos. El Portal podría tener una base con usuarios, pedidos, etc.; Stock su base con productos, inventarios; Transporte su base con envíos, estados. Cuando se necesita información de otro dominio, se hace vía API en lugar de consultas directas a la base ajena. Esto sigue el principio de *encapsulamiento de datos*. (En una etapa inicial, a veces se usan bases compartidas por simplicidad, pero entonces realmente no están desacoplados los servicios).
- **Manejabilidad y monitoreo:** A nivel práctico, tener microservicios implica más piezas desplegadas. Es recomendable contar con logs claros en cada servicio y quizá centralizados (para poder seguir el rastro de una operación que involucra varios servicios). También manejar el versionado de APIs para futuras extensiones sin romper compatibilidad (por ej., URL base /api/v1/ vs /api/v2/ cuando hagamos cambios mayores).
- **Escalabilidad:** La ventaja de microservicios es que puedes **escalar independientemente**. Si el servicio de Portal recibe muchas más solicitudes que el de Stock, puedes desplegar más instancias de Portal sin tocar Stock. Piensa en ello para distribuir la carga. En nuestro caso de estudio, posiblemente el Portal web será el más concurrido, mientras que Stock (usado solo por vendedores) tiene menos carga, y Transporte quizás intermedia. Arquitectura en la nube con contenedores (Docker) y orquestadores (Kubernetes) suele usarse para manejar esto, aunque para un proyecto académico podríamos simularlo con procesos separados.
- **Tolerancia a fallos:** ¿Qué ocurre si el servicio de Stock se cae en pleno proceso de compra? El Portal debería manejar esa situación (por ejemplo, dando un mensaje de error al usuario "No se pudo confirmar stock, intenta más tarde") en lugar de quedar bloqueado indefinidamente. Implementar **timeouts** en las llamadas inter-servicios y planes de recuperación es importante. A veces se usan *circuit breakers* – librerías que evitan intentar conectar repetidamente a un servicio caído por un tiempo. Esto tal vez exceda el alcance de su proyecto, pero al menos considerar las fallas: cada servicio debería fallar de forma que no colapse todo el sistema. Por ejemplo, si Logística falla, el Portal aún debería permitir compras (quedando pendiente el cálculo de envío o haciendo un estimado por defecto).
- **Consistencia de la configuración:** Asegúrense de mantener la **configuración consistente** entre servicios (por ejemplo, URLs de los endpoints de cada servicio quizás conviene ponerlos en un archivo de config o variable de entorno para no hardcodear). También todos deberían usar HTTPS si se exponen externamente. Y manejar CORS adecuadamente si por ejemplo el front-end (Portal) está en un dominio y hace peticiones a Stock/Transporte en otros dominios.

En conclusión, diseñar estos microservicios implica pensar en **cómo dividir el problema** (lo cual el enunciado ya les da) y en **cómo se comunican**. Pueden utilizar Django/DRF para construir cada servicio con su propia base de datos y API. Algunos estudiantes implementan todo en un solo proyecto Django por simplicidad, usando apps para separar las áreas, pero la verdadera arquitectura microservicios implicaría proyectos desplegados por separado. Quizá, dependiendo de los recursos, podrían simular microservicios corriendo en distintos puertos o usando Docker para contener cada uno.

Lo más importante es mantener un buen **contrato de integración**: definan bien los endpoints que unos servicios necesitan de otros. Por ejemplo: - El Portal necesitará un endpoint en Stock para "reservar stock" de X producto (disminuyendo stock disponible temporalmente). - El Portal necesitará un endpoint en Transporte para "crear un envío" (pasándole datos del pedido). - Transporte necesitará un endpoint en Stock para "obtener detalles de productos por ID" (peso, dimensiones, etc. para cálculo de envío). - Transporte (o una tarea programada) necesitará un mecanismo para "liberar stock" en Stock si

un envío fue cancelado/no concretado. - Transporte/Stock podrían necesitar notificar al Portal de ciertos eventos (entrega realizada, stock insuficiente, etc.).

Si diseñan esas interfaces con antelación (quizá usando OpenAPI spec), la implementación será más clara y cada equipo puede trabajar en paralelo en su servicio, simulando las respuestas esperadas de los demás hasta integrarlos.

7. Recursos útiles (documentación, ayuda y ejemplos)

Al afrontar este proyecto, es fundamental apoyarse en la documentación oficial y en la comunidad de desarrolladores. A continuación, se listan recursos y referencias recomendadas:

- **Documentación oficial de Django:** Disponible en docs.djangoproject.com. Ofrece tutoriales de inicio, guía de temas avanzados y referencias de API de todos los componentes de Django. En particular, el tutorial oficial de Django (parte 1 a 7) puede ser útil si aún no están familiarizados con el framework web en sí.
- **Documentación oficial de Django REST Framework:** Sitio django-rest-framework.org – Incluye un tutorial rápido, guía de serializers, views, authentication, etc. Es altamente recomendable leer el *Quickstart* y la sección de *API Guide* para entender cómo piensa DRF. (La comunidad de DRF es activa, y hay muchas preguntas respondidas en Stack Overflow también).
- **MDN Web Docs (Mozilla Developer Network):** developer.mozilla.org – Excelente documentación sobre tecnologías web. Por ejemplo, para repasar conceptos de HTTP (métodos, códigos de estado, cabeceras), MDN es muy claro. Está disponible en español e inglés. Si necesitas entender qué significa cierto código de error, o cómo funcionan las cookies, CORS, etc., aquí lo encontrarás.
- **DevDocs.io:** devdocs.io – Un portal que agrupa documentación de múltiples tecnologías (incluyendo Django, Python, MDN, etc.) con búsqueda rápida. Es muy útil tenerlo abierto mientras codificas para buscar detalles de alguna función o componente sin navegar entre sitios.
- **Stack Overflow:** stackoverflow.com – El foro de preguntas y respuestas de programadores. Seguramente cualquier error que te encuentres o duda específica (por ejemplo "TypeError: X is not JSON serializable" o "Django import error ...") alguien ya la preguntó antes. Busca en Google agregando "Stack Overflow" o directamente en su buscador. Ten en cuenta marcar bien la versión de Django/DRF que estás usando si preguntas algo. *(Un tip: en muchos errores de Django, leer el* traceback *y mensaje detenidamente ya da pistas, pero StackOverflow ayuda con soluciones concretas de otros.)*
- **Documentación de JWT:** El sitio oficial jwt.io – Incluye información básica, ejemplos de cómo se componen los tokens e incluso una herramienta para decodificar tokens (¡útil para depuración, pero nunca pegues tu token de producción ahí por seguridad!). También la especificación RFC 7519 si alguien quiere detalles formales.
- **Blogs y tutoriales:** Existen innumerables blogs con tutoriales de Django, DRF, JWT, etc. Algunos en español como *SóloPython*, *Django Girls (tutorial)*, o en inglés *SimpleIsBetterThanComplex*, *TestDriven.io*, etc., que explican desde lo básico a casos avanzados. Úsenlos para diferentes perspectivas. (Buscar "Tutorial Django Rest Framework español" en Google/YouTube puede darles guías paso a paso similares a lo que hicimos).
- **Comunidades y foros en español:** Si prefieren discusiones en español, foros como Foros de Python en español o comunidades en Telegram/Discord de Python/Django pueden ser de ayuda. Incluso la comunidad de UTN o grupos de alumnos de años anteriores podrían tener apuntes.
- **Herramientas de prueba de APIs:** Postman (interfaz gráfica) o Insomnia son muy útiles para ir testeando sus APIs con distintas peticiones, guardar colecciones de request, etc. Postman incluso tiene capacidad de generar documentación y tests. Alternativamente, usar *httpie* o *curl*

en la terminal como vimos también es válido. Lo importante es probar cada endpoint exhaustivamente.

- **GitHub y ejemplos de proyectos:** Puede ser instructivo revisar proyectos existentes. En GitHub hay muchos repositorios de ejemplo con Django REST, o incluso específicos a e-commerce. Busquen por "Django REST e-commerce" (aunque cuidado de no abrumarse, pueden ser avanzados). Leer código ajeno a veces ayuda a encontrar soluciones o enfoques diferentes.
- **UML y diagramas:** Dado que este es un proyecto integrador, puede que necesiten también hacer diagramas de arquitectura, de clases, de secuencia, etc. Herramientas como Draw.io (gratuita online) o incluso a mano alzada les servirán para planificar. En los PDF proporcionados (UML manual de usuario/referencia) tienen la teoría para hacer diagramas UML si les requieren en el informe.

En general, el enfoque debe ser: *leer documentación oficial primero*, luego apoyarse en foros/comunidad para dudas específicas o problemas, y buscar ejemplos prácticos para orientar la implementación. No teman consultar varias fuentes; por ejemplo, la documentación oficial de DRF es completa pero puede sentirse densa, entonces un tutorial paso a paso de un blog puede complementarla para entender mejor.

8. Videos de YouTube recomendados (por tema)

Para reforzar cada concepto, aquí se listan algunos videos útiles (en español) que explican o muestran de forma práctica los temas tratados:

- **¿Qué es una API?** – *"¿Qué es una API? - La mejor explicación en español"* (Video de EDteam, 9 min). Un video corto y claro que explica con analogías qué son las APIs y por qué son útiles, ideal para principiantes. También: *"¿Qué es una API? Explicado con Ejemplos Claros y Simples"* (YouTube, 12 min) brinda ejemplos visuales sencillos.
- **¿Qué es una API REST?** – *"¿Qué es una REST API y cómo funciona?"* (Video, ~12 min, canal UskoKruM2010). Explica los fundamentos de REST con ejemplos ilustrativos. También EDteam tiene una charla larga sobre REST. Estos videos te ayudarán a afianzar conceptos como recursos, métodos, etc., con explicaciones visuales.
- **Django y Django REST Framework (Tutorial):** – *"Crea una REST API (GET, POST, PUT, DELETE) | Tutorial desde Cero"* (Video, ~43 min). Un excelente recorrido práctico creando una API REST con Django REST Framework, cubriendo la instalación, serializadores, viewsets y probando las operaciones CRUD. Perfecto si quieres ver el proceso completo en acción. Para algo más extenso: *"Curso Django REST Framework - Proyecto desde CERO"* (serie en YouTube) profundiza en varios aspectos y buenas prácticas.
- **JWT y autenticación en Django REST:** – *"Proteger API en Django Rest Framework con JWT"* (Video corto, ~11 min). Muestra paso a paso cómo implementar JWT en DRF usando SimpleJWT, muy alineado con lo que describimos en este tutorial. Complementario: *"Autenticación con JWT (JSON Web Tokens) en REST API"* (OpenWebinars, 30 min) para entender a fondo JWT (no específico de Django, pero conceptual).
- **Documentación OpenAPI/Swagger:** – *"Aprende a documentar tu API con Swagger"* (Video, ~37 min, canal MoonCode). Explica qué es Swagger/OpenAPI y demuestra cómo documentar una API existente, incluyendo la configuración de Swagger UI. Aunque su ejemplo no es Django sino Node, los conceptos son trasladables (y hay otros videos específicos de Django + Swagger en inglés si te animas).
- **Microservicios – Arquitectura y Ejemplos:** – *"¿Qué son y cómo funcionan los microservicios?"* (Video de EDteam, ~42 min). Una charla que introduce la arquitectura de microservicios, por qué surgió, casos de uso (ej: Netflix), pros y contras. Muy útil para entender el panorama general. También *"Arquitectura de Microservicios: una mirada teórico-práctica"* (YouTube) cubre patrones de

diseño de microservicios con ejemplos. Si buscas algo específico a e-commerce: *"Microservicios en e-commerce"* (ITDO, webinar) podría ser interesante.

- **Extra - Buenas prácticas y consejos:** – *"13 mejores prácticas para asegurar microservicios"* (Geekflare, inglés con subtítulos) y *"Patrones de diseño en Microservicios"* (Paradigma, español) son videos que, si bien más avanzados, pueden darte ideas de cómo abordar ciertas cosas (como comunicación asíncrona, seguridad, etc.). Úsalos si quieres profundizar en arquitectura.

Cada uno de estos videos complementa la teoría con demostraciones o explicaciones auditivas/visuales, lo cual puede ser muy beneficioso para afianzar el conocimiento. ¡Aprovecha el contenido gratuito de calidad que existe!

Conclusión: Con este tutorial has recibido una introducción integral a los temas clave para encarar el proyecto: sabes qué es una API y los métodos HTTP básicos; entiendes el estilo REST y cómo diseñar recursos; aprendiste a usar Django REST Framework para construir rápidamente una API REST (y probablemente querrás seguir practicando para afianzarlo); comprendiste cómo funciona JWT para autenticación stateless en APIs y cómo implementarlo en Django; viste la importancia de documentar adecuadamente la API con OpenAPI/Swagger; y discutimos la arquitectura propuesta de microservicios para la plataforma de compras, con buenas prácticas para su comunicación. Tienes además referencias y videos para profundizar cada punto.

El siguiente paso es aplicar este conocimiento en el desarrollo real: comienza creando los servicios básicos, prueba las interacciones, y ve iterando. No te desanimes ante los obstáculos (bugs, configuraciones) – con la documentación y la comunidad a tu disposición, cualquier problema tiene solución. ¡Mucho éxito con tu proyecto de plataforma de compras online!

---

[1] API - Wikipedia, la enciclopedia libre
http://es.wikipedia.org/wiki/Interfaz_de_programaci%C3%B3n_de_aplicaciones

[2] [3] ¿Cómo diseñar una API? Introducción a OpenAPI Specification | Guillermo Alvarado
https://galvarado.com.mx/post/apidesign/