ChatGPT

Tutorial: Documentación de APIs Django con OpenAPI (YAML)

Introducción: OpenAPI y su relación con Swagger

OpenAPI es una especificación estándar para describir APIs RESTful de forma independiente del lenguaje. Permite definir en un formato legible (YAML o JSON) todos los **endpoints**, métodos HTTP, parámetros, estructuras de datos y respuestas de nuestra API. Originalmente, esta especificación nació con el nombre *Swagger Specification*, creada por SmartBear en 2011, y luego fue donada en 2015 a la OpenAPI Initiative, que la renombró a **OpenAPI Specification (OAS)** [1] [2]. Por ello, hoy en día **Swagger** suele referirse al **conjunto de herramientas** (Swagger UI, Swagger Editor, etc.) que trabajan con la especificación OpenAPI, mientras que **OpenAPI** hace referencia al **formato estándar** en sí mismo [3]. En resumen, OpenAPI define **cómo describir** una API REST, y Swagger provee herramientas para **visualizar, probar y generar** documentación a partir de esa descripción.

¿Por qué documentar manualmente una API Django con OpenAPI? A pesar de que frameworks como Django REST Framework ofrecen generación automática de esquemas, hacerlo manualmente nos da un control total sobre la documentación. Podemos detallar cada caso, asegurar que la especificación esté siempre actualizada y compartirla con desarrolladores front-end o terceros. Además, una especificación OpenAPI bien hecha puede ser importada en herramientas como Swagger UI, Postman o Swagger Codegen para generar clientes y probar la API fácilmente.

En este tutorial paso a paso, crearemos un archivo YAML de OpenAPI **desde cero** para documentar una API Django típica con endpoints de autenticación JWT. Cubriremos la estructura básica del archivo, cómo documentar endpoints comunes (registro, login, verificación de token, perfil de usuario), cómo definir **componentes reutilizables** (esquemas de datos, respuestas de error, seguridad JWT) y finalmente cómo consumir la API desde un cliente web usando el token JWT. ¡Comencemos!

Paso 1: Estructura básica de un archivo OpenAPI en YAML

Una definición OpenAPI es esencialmente un documento (en YAML o JSON) que sigue un esquema predefinido. Los elementos principales de un archivo OpenAPI 3.x incluyen:

- **openapi:** la versión de la especificación que usamos (por ejemplo, "3.0.3" o "3.1.0").
- **info:** metadatos de la API (título, versión, descripción, contacto, etc.).
- **servers:** la(s) URL base donde se aloja la API (por ejemplo, desarrollo, producción).
- **paths:** la lista de endpoints (rutas) de la API y sus operaciones (GET, POST, etc.).
- **components:** definiciones reutilizables (esquemas de datos, parámetros, respuestas, esquemas de seguridad, etc.).
- **security:** configuración de autenticación global (opcional, puede definirse globalmente o por operación).

Según la especificación OpenAPI 3, nuestro documento debe contener al menos `openapi`, `info` y `paths` [4]. Empecemos definiendo la cabecera básica de nuestro archivo YAML con la versión, la info y un servidor de ejemplo:

```
openapi: "3.0.3"  # Versión de la especificación OpenAPI que se utiliza
info:
  title: "API de ejemplo - Tutorial OpenAPI"   # Título de la API
  version: "1.0.0"                             # Versión de la API
  description: |
    Documentación de la API de ejemplo, escrita manualmente en formato OpenAPI
3.
    Esta API incluye autenticación JWT y endpoints de usuario.
servers:
  - url: "http://localhost:8000/api"           # URL base de la API (ejemplo
entorno local)
    description: "Servidor de desarrollo local"
```

Explicación: En la cabecera hemos especificado la versión de OpenAPI (`3.0.3`). Bajo `info` proporcionamos un título descriptivo, un número de versión de nuestra API y una descripción general (el símbolo `|` permite escribir descripciones en bloque multilínea en YAML). En `servers` listamos la URL base de nuestro servicio; aquí usamos un servidor local típico de Django (`localhost:8000/api`), pero podríamos añadir más (por ejemplo, uno para producción). Estos servidores aparecerán en la documentación generada, facilitando probar la API en diferentes entornos.

> **Nota:** Es posible incluir más metadatos en `info` como términos de servicio, contacto, licencias, etc., pero para este tutorial mantenemos solo los campos esenciales. La sección `servers` también puede listar múltiples entornos (desarrollo, staging, producción). Si se omite `servers`, por defecto muchas herramientas asumen `localhost` en la documentación.

Con la estructura base lista, pasemos a documentar los endpoints en la sección `paths`. Inicialmente, nuestra sección `paths` estará vacía, pero la iremos llenando en los siguientes pasos.

Paso 2: Documentar el endpoint de Registro de usuario

Comencemos con un endpoint fundamental: **registro de nuevos usuarios**. Imaginemos que nuestra API tiene una ruta `POST /auth/register` donde un cliente puede enviar los datos de un nuevo usuario (p. ej., nombre de usuario, email y contraseña) para crear una cuenta. Este endpoint no requiere autenticación previa (es abierto) y al ejecutarse correctamente creará un nuevo usuario.

Para documentar este endpoint en OpenAPI, agregaremos una entrada bajo `paths` con la ruta `/auth/register` y dentro definiremos el método `post` (ya que el registro suele ser vía POST). Debemos especificar detalles como resumen, descripción, los datos que espera en la petición (request body) y las posibles respuestas.

Añadamos entonces en nuestro YAML:

```
paths:
  /auth/register:
    post:
      summary: "Registro de nuevo usuario"
      description: "Permite registrar un nuevo usuario proporcionando nombre de
usuario, email y contraseña."
      tags:
        - "Autenticación"
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/RegistroUsuarioEntrada"
            example:
              username: "nuevo_usuario"
              email: "usuario@example.com"
              password: "ContraseñaSegura123"
      responses:
        "201":
          description: "Usuario creado exitosamente."
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/UsuarioDetalle"
        "400":
          description: "Solicitud inválida. Por ejemplo, datos faltantes o
formato de email incorrecto."
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/ErrorRespuesta"
```

Explicación: Hemos documentado la ruta `/auth/register` con el método **POST**. Veamos cada parte:

- `summary` y `description` ofrecen un resumen breve y una descripción más detallada de la operación. Aquí aclaramos que crea un nuevo usuario con los datos proporcionados.
- `tags` nos permite agrupar la operación en categorías; usamos la etiqueta "Autenticación" para todos los endpoints relacionados al proceso de autenticación (registro, login, etc.). Esto es opcional pero útil para organizar la documentación.
- `requestBody` describe el cuerpo de la petición que este endpoint espera. Marcamos `required: true` porque el registro requiere obligatoriamente un JSON con los datos del nuevo usuario. Bajo `content` especificamos que el cuerpo debe ser `application/json` e indicamos su esquema.

- Usamos `$ref: "#/components/schemas/RegistroUsuarioEntrada"` para referenciar un **esquema reutilizable** (que definiremos más adelante en `components.schemas`). Este esquema representará la estructura JSON esperada en el registro (por ejemplo, contendrá `username`, `email`, `password`).
- Además, proporcionamos un `example` de dicho JSON para mayor claridad en la documentación (un usuario y contraseña de ejemplo). Esto ayuda a consumidores a entender el formato esperado.
- `responses` lista las posibles respuestas que el endpoint puede dar.
- La respuesta **"201" (Created)** indica que el usuario se creó correctamente. Incluimos una breve descripción y especificamos que la respuesta tendrá contenido JSON con el esquema `$ref: "#/components/schemas/UsuarioDetalle"`. Este esquema (que también definiremos más adelante) representará los datos del usuario creado (por ejemplo, su `id`, `username`, `email`, etc.) que la API devuelve tras el registro. Muchas APIs pueden devolver el objeto creado o algún mensaje de éxito; aquí asumiremos que devuelve los datos básicos del nuevo usuario.
- La respuesta **"400" (Bad Request)** cubre el caso de error cuando la solicitud es incorrecta, por ejemplo si faltan campos obligatorios o el email tiene un formato inválido. Indicamos que la respuesta tendrá una estructura de error común, referenciada como `$ref: "#/components/schemas/ErrorRespuesta"`. Usaremos este esquema genérico de error para documentar mensajes de error consistentes (lo definiremos en la sección de componentes). La descripción aclara la situación (solicitud inválida por datos erróneos).

Notar que hemos introducido referencias (`$ref`) a esquemas **no definidos aún**: `RegistroUsuarioEntrada`, `UsuarioDetalle` y `ErrorRespuesta`. No te preocupes, en el **Paso 6** crearemos estos esquemas en la sección `components` para formalizar la estructura de los datos de entrada/salida y errores. Por ahora, hemos descrito el endpoint de registro asumiendo esos componentes.

> **¿Por qué usar `$ref` para esquemas?** Esto hace nuestra especificación más limpia y evita repetir la estructura de objetos en múltiples lugares. Definimos el esquema una vez (por ejemplo, cómo luce un objeto Usuario) y luego lo reutilizamos en varias respuestas o peticiones. Esta práctica sigue el principio DRY (Don't Repeat Yourself) [5].

Paso 3: Documentar el endpoint de Login con JWT

Otro endpoint típico es **login** de usuario, donde el cliente envía sus credenciales y la API responde con un token JWT para autenticar futuras solicitudes. Supondremos una ruta `POST /auth/login` que recibe nombre de usuario y contraseña, y devuelve un token JWT si las credenciales son correctas.

Documentemos `/auth/login` similar al anterior:

```
  /auth/login:
    post:
      summary: "Autenticación de usuario (Login)"
      description: "Verifica las credenciales del usuario y devuelve un token
JWT para futuras peticiones."
      tags:
        - "Autenticación"
      requestBody:
```

```
        required: true
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/LoginEntrada"
            example:
              username: "nuevo_usuario"
              password: "ContraseñaSegura123"
      responses:
        "200":
          description: "Login exitoso. Devuelve el token de acceso JWT."
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/LoginRespuesta"
        "401":
          description: "Credenciales inválidas. El usuario/contraseña no son
correctos."
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/ErrorRespuesta"
```

Explicación: Aquí definimos el endpoint `POST /auth/login`:

- En `requestBody`, esperamos un JSON con las credenciales. Referenciamos un esquema `LoginEntrada` que contendrá típicamente el `username` y `password` del usuario (y, según el caso, podría ser email y contraseña; en nuestro ejemplo usamos username). Incluimos un ejemplo JSON de credenciales.
- Para la respuesta de éxito `200`, indicamos que la API devuelve un token JWT. Definimos un esquema de respuesta `LoginRespuesta` que representará un objeto con el token (por ejemplo, `{ "token": "eyJhbGciOiJIUzI1..." }`). Muchas APIs simplemente devuelven el token, aunque otras pueden incluir información adicional como fecha de expiración o datos del usuario; adaptaremos el esquema según nuestras necesidades en la sección de componentes.
- En caso de credenciales inválidas, documentamos una respuesta `401 Unauthorized`. La descripción aclara que usuario o contraseña no son correctos, y de nuevo usamos el esquema de error genérico `ErrorRespuesta` para el cuerpo de la respuesta (por ejemplo, podría devolver `{"detail": "Credenciales inválidas"}`). Un **401** es apropiado cuando las credenciales proporcionadas no autorizan el acceso.

  Nota: También podríamos documentar un código **400 Bad Request** si, por ejemplo, faltan campos en la solicitud de login (similar al caso del registro). Para simplificar, hemos puesto solo el caso de credenciales inválidas como 401. En una documentación completa, podrías incluir ambos. Lo importante es enumerar las respuestas relevantes para que el consumidor de la API sepa manejarlas.

Tras el login exitoso, el cliente obtendrá un token JWT que deberá enviar en las siguientes peticiones protegidas. Más adelante explicaremos cómo se documenta el uso de JWT en los headers con OpenAPI (ver **Paso 7**).

Paso 4: Documentar el endpoint de Verificación de token JWT

Supongamos que nuestra API ofrece un endpoint para **verificar la validez de un token JWT**. Esto es común cuando se usa JWT: por ejemplo, Django REST Framework SimpleJWT provee un endpoint `/auth/token/verify/` donde el cliente envía un token y la API responde si es válido o no. Aunque no todas las implementaciones requieren esto (a veces basta con usar el token directamente en endpoints protegidos), lo incluiremos como ejemplo educativo.

Imaginemos entonces una ruta `POST /auth/token/verify` que recibe un token (en el cuerpo de la petición) y devuelve una confirmación de validez. Si el token es válido, podríamos devolver un 200 OK con algún mensaje; si es inválido o expiró, devolver un error 401.

Documentemos este endpoint:

```
  /auth/token/verify:
    post:
      summary: "Verificar validez de un token JWT"
      description: "Recibe un token JWT y comprueba si sigue siendo válido (no
expirado, firma correcta)."
      tags:
        - "Autenticación"
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/VerificarTokenEntrada"
            example:
              token: "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."  # Token JWT de
ejemplo
      responses:
        "200":
          description: "Token válido. El token JWT proporcionado es auténtico y
no ha expirado."
          content:
            application/json:
              schema:
                type: object
                properties:
                  detail:
                    type: string
                    example: "Token válido."
```

```
        "401":
          description: "Token inválido o expirado. El token proporcionado no es
válido."
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/ErrorRespuesta"
```

Explicación: En `/auth/token/verify` definimos:

- En `requestBody`, esperamos un JSON con el token JWT. Referenciamos un esquema `VerificarTokenEntrada` (que básicamente tendrá un solo campo `token` de tipo string). Proporcionamos un ejemplo de token JWT (truncado para brevedad) en el campo `example`.
- La respuesta "200" indica que el token es válido. Aquí, para demostrar, incluimos un cuerpo con un simple mensaje `{ "detail": "Token válido." }`. Algunos endpoints de verificación podrían no devolver ningún contenido (solo un 200 OK vacío) o podrían devolver datos decodificados del token. Nosotros optamos por un mensaje de confirmación para ilustrar cómo documentar una respuesta de éxito con mensaje. Observa que aquí no usamos `$ref` a un esquema porque es una respuesta específica de este endpoint. Podríamos haber definido un esquema de mensaje genérico, pero para simplicidad lo mostramos en línea.
- La respuesta "401" cubre el caso de token inválido o expirado. Reutilizamos `ErrorRespuesta` para indicar el error (p.ej., podría devolver `{"detail": "Token inválido"}`).

Con esto, el endpoint de verificación queda documentado. Recapitulando: este endpoint no requiere autenticación por header (ya que él mismo está verificando un token), sino que recibe el token en el cuerpo. En cambio, el próximo endpoint **sí** requerirá el token en el header, lo que nos lleva a ver cómo documentar la seguridad JWT.

Paso 5: Documentar el endpoint de Perfil de usuario autenticado

Finalmente, un endpoint muy común en APIs protegidas es obtener el **perfil del usuario autenticado**. Por ejemplo, `GET /users/me` podría retornar los datos del usuario actualmente logueado (basado en el JWT enviado). Este endpoint **requiere autenticación** mediante el token JWT en la cabecera de la petición.

Supondremos una ruta `GET /users/me` que devuelve la información del perfil (por ejemplo, id, nombre de usuario, email, etc. del usuario dueño del token). Veamos cómo documentarlo:

```
  /users/me:
    get:
      summary: "Obtener perfil del usuario autenticado"
      description: "Retorna los datos de perfil del usuario que ha iniciado
sesión (según el JWT proporcionado)."
      tags:
        - "Usuarios"
      security:
        - bearerAuth: []
```

```
      responses:
        "200":
          description: "Perfil del usuario obtenido con éxito."
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/UsuarioDetalle"
        "401":
          description: "No autorizado. El token JWT no fue enviado o es
inválido."
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/ErrorRespuesta"
```

Explicación: La ruta `/users/me` con método GET representa la obtención del perfil del usuario autenticado:

- Hemos etiquetado este endpoint bajo `tags: ["Usuarios"]` para separarlo de los de autenticación. (Esto es arbitrario; podríamos seguir con "Autenticación", pero elegimos "Usuarios" para indicar que es parte de operaciones de usuario).
- Lo **más importante** aquí es la sección `security`. Al incluir `security: - bearerAuth: []`, estamos declarando que esta operación requiere autenticación mediante el esquema de seguridad `bearerAuth`. En OpenAPI 3, `security` es una lista de *esquemas de seguridad* que aplican a la operación; cada elemento es un objeto con claves como `bearerAuth` y valor una lista de alcances (para OAuth2, en JWT no se usan scopes típicamente, así que ponemos una lista vacía). En esencia, esta línea significa: "*Para llamar a GET /users/me, el cliente debe proporcionar un token válido del tipo Bearer*". Aún no hemos definido qué es `bearerAuth` – eso lo haremos en la sección de componentes de seguridad – pero ya lo referenciamos aquí. Esta es la forma estándar en OpenAPI de requerir JWT u otra auth: se define un esquema de seguridad global y luego se referencia en los endpoints que lo necesitan [6].
- Las respuestas:
- "200" devuelve el perfil del usuario. Usamos nuevamente el esquema `UsuarioDetalle` para estructurar el JSON de salida (el mismo que en el registro, asumiendo que tanto al registrarse como al consultar perfil se devuelven datos del usuario). Por ejemplo, contendrá el `id`, `username`, `email`, etc. (Veremos su definición enseguida).
- "401 Unauthorized" cubre el caso en que no se envía token, este es inválido, o expiró. La descripción indica *No autorizado* y se reutiliza `ErrorRespuesta` para el mensaje de error. En la práctica, si el token falta o es inválido, Django REST Framework devolverá un 401 con un detalle como "Credenciales no provistas" o "Token inválido", que encaja en este esquema de error.

Con esto, hemos documentado los cuatro endpoints solicitados: registro, login, verificación de token, y perfil de usuario. Observa cómo empleamos los **componentes reutilizables** (`RegistroUsuarioEntrada`, `LoginEntrada`, `LoginRespuesta`, `UsuarioDetalle`, `ErrorRespuesta`, y también `bearerAuth` para la seguridad) en varias partes de la especificación. En el siguiente paso, vamos a definir todos esos componentes bajo la sección `components` para completar nuestra especificación.

Paso 6: Definir componentes reutilizables (esquemas de datos y errores)

La sección `components` del archivo OpenAPI nos permite centralizar definiciones de **esquemas** (modelos de datos), **respuestas**, **parámetros**, **securitySchemes**, etc., de modo que puedan referenciarse desde las operaciones anteriores. Ya en nuestros endpoints usamos `$ref` apuntando a `#/components/schemas/...` para varios elementos. Ahora vamos a crear esas definiciones.

Agreguemos al YAML una sección `components` con los esquemas mencionados:

```yaml
components:
  schemas:
    # Esquema de los datos que se envían para registrar un nuevo usuario
    RegistroUsuarioEntrada:
      type: object
      required:
        - username
        - email
        - password
      properties:
        username:
          type: string
          example: "nuevo_usuario"
        email:
          type: string
          format: email
          example: "usuario@example.com"
        password:
          type: string
          format: password
          example: "ContraseñaSegura123"
      description: "Campos requeridos para registrar un nuevo usuario."

    # Esquema de los datos que se envían para hacer login
    LoginEntrada:
      type: object
      required:
        - username
        - password
      properties:
        username:
          type: string
          example: "nuevo_usuario"
        password:
          type: string
          format: password
```

```
          example: "ContraseñaSegura123"
      description: "Credenciales requeridas para login (usuario y contraseña)."

    # Esquema de la respuesta exitosa al hacer login (token JWT)
    LoginRespuesta:
      type: object
      properties:
        token:
          type: string
          description: "Token JWT de autenticación. Usar este valor en el
header Authorization."
          example: "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9..."  # Token truncado
      description: "Respuesta devuelta tras login exitoso, contiene el token
JWT para autenticación."

    # Esquema de los datos que se envían para verificar un token (token JWT en
el cuerpo)
    VerificarTokenEntrada:
      type: object
      required:
        - token
      properties:
        token:
          type: string
          description: "JWT a verificar"
          example: "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."  # Token truncado
      description: "Estructura de la petición para verificar la validez de un
token JWT."

    # Esquema de la estructura de un usuario (datos de perfil) devueltos por la
API
    UsuarioDetalle:
      type: object
      properties:
        id:
          type: integer
          example: 123
        username:
          type: string
          example: "nuevo_usuario"
        email:
          type: string
          example: "usuario@example.com"
        # Aquí podrían incluirse más campos de perfil, e.g. nombre, apellidos,
etc.
      description: "Datos de perfil del usuario."

    # Esquema genérico para respuestas de error (mensaje de error)
```

```
    ErrorRespuesta:
      type: object
      properties:
        detail:
          type: string
          example: "Error: credenciales inválidas."
      description: "Formato estándar de respuesta de error con un mensaje en el
campo 'detail'."
```

Explicación de los esquemas definidos:

- **RegistroUsuarioEntrada:** representa el cuerpo JSON requerido para registrar un usuario. Es un objeto con `username`, `email` y `password`. Marcamos los tres campos como `required`. Usamos `format: email` para el email (OpenAPI reconoce formatos como email, uri, date-time, etc.), y `format: password` para la contraseña (esto indica a herramientas como Swagger UI que pueden ocultar el input). Cada campo tiene un ejemplo.

- **LoginEntrada:** estructura similar a la anterior, pero solo con `username` y `password` (campos requeridos para autenticarse).

- **LoginRespuesta:** describe el JSON que devolvemos al hacer login exitoso. Aquí definimos un objeto con un campo `token` de tipo string. En la descripción aclaramos que es un token JWT de autenticación que se usará en el header *Authorization*. Proporcionamos un ejemplo (truncado por longitud). Si tu API devuelve también un token de refresco u otros datos, podrías extender este esquema (p.ej., agregar `refreshToken` o datos del usuario). Para nuestro caso simple, solo devolvemos un token de acceso.

- **VerificarTokenEntrada:** un esquema sencillo para el cuerpo de la petición de verificación de token, que solo contiene el `token` (string requerido) a verificar.

- **UsuarioDetalle:** describe la respuesta con los datos de un usuario. Incluimos `id`, `username`, `email` como propiedades de ejemplo. Puedes agregar otros campos que tu API exponga (por ejemplo, nombre completo, fecha de registro, etc.). Este esquema lo usamos tanto en la respuesta 201 de registro como en la respuesta 200 de perfil de usuario, asegurando consistencia en la documentación de qué campos tiene un usuario.

- **ErrorRespuesta:** esquema estándar de error. Aquí optamos por un objeto con un único campo `detail` de tipo string, que contiene un mensaje de error. Django REST Framework por defecto suele enviar errores en esta forma (por ejemplo, `{"detail": "Las credenciales no son válidas."}` para 401, o mensajes similares). Si tu API utiliza otro formato (p.ej. `{"error": "...", "code": ...}` o errores campo-por-campo), puedes ajustar este esquema o crear varios. Para simplificar, usamos un formato genérico con `detail`. Esto se referenció en las respuestas 400/401 de nuestros endpoints.

Ahora hemos definido todos los **schemas** referenciados. Recuerda que estos se encuentran bajo `components/schemas` en el YAML, y los referenciamos con `#/components/schemas/NombreEsquema` en otros lugares.

Paso 7: Configurar la seguridad JWT con `securitySchemes`

En OpenAPI, la autenticación y autorización se describen mediante **Security Schemes**. Dado que nuestra API utiliza JWT (JSON Web Tokens) como método de autenticación, configuraremos un esquema de seguridad tipo **Bearer token**. Esto le indicará a herramientas como Swagger UI cómo proveer el token en las peticiones.

Agregaremos dentro de `components` una subsección `securitySchemes` para definir nuestro esquema JWT:

```
  securitySchemes:
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
      description: "Autenticación JWT usando el esquema Bearer. Incluya el
token como 'Bearer <JWT>' en el header Authorization."
```

Explicación: Aquí definimos un esquema de seguridad llamado `bearerAuth` (el nombre es arbitrario, pero elegimos uno descriptivo). Desglosando los campos:

- `type: http` indica que es un esquema de autenticación HTTP (OpenAPI también soporta `apiKey`, `oauth2`, `openIdConnect` como otros tipos).
- `scheme: bearer` indica el tipo de *HTTP Authentication* utilizado es Bearer Token (portador). Este es el esquema estándar para tokens.
- `bearerFormat: JWT` sugiere que el formato del token es un JWT específico (JSON Web Token). Aunque técnicamente la autenticación Bearer puede usar tokens opacos, especificar JWT ayuda a clarificar la intención. Es informativo para humanos y algunas herramientas.
- `description` opcionalmente describe cómo funciona este esquema. Indicamos que se debe incluir el token con el prefijo "Bearer " en el header **Authorization** de las peticiones. Esto es exactamente lo que hará un cliente: enviar `Authorization: Bearer <token_jwt>`.

Al definir `bearerAuth` aquí, todas las operaciones que en su apartado `security` incluyan `bearerAuth: []` requerirán que el consumidor provea un token. En Swagger UI, por ejemplo, aparecerá un botón "*Authorize*" que permitirá ingresar el token JWT, el cual luego se añadirá a las solicitudes automáticamente.

Nuestra especificación ya usa este esquema en el endpoint `/users/me` (vía `security: - bearerAuth: []`). Si quisiéramos, podríamos aplicar el requisito de seguridad globalmente a toda la API colocando un nodo `security` al nivel superior (así todos los endpoints requerirían auth por defecto, salvo los que se exceptúen). En este tutorial preferimos definirlo solo en los

endpoints que lo necesitan, dado que **/auth/register**, **/auth/login** y **/auth/token/verify** deben estar accesibles sin token.

Para verificar que todo está conectado: el cliente deberá obtener el token del login (como documentamos en `LoginRespuesta`) y luego incluirlo en la cabecera `Authorization` para acceder, por ejemplo, a `/users/me`. Nuestra documentación ahora refleja claramente este flujo: el esquema de seguridad JWT está definido y aplicado donde corresponde [7] [8].

Paso 8: Consumir la API desde un cliente web utilizando JWT

Ahora que tenemos documentada la API, veamos **cómo un cliente web consumiría esta API usando JWT**. Explicaremos brevemente el flujo utilizando *fetch* y *axios* (bibliotecas comunes en JavaScript para peticiones HTTP):

1. **Registro/Login para obtener el JWT:** El cliente primero usa el endpoint de **login** (o registra un usuario nuevo) para obtener el token. Por ejemplo, con *fetch* podría hacer:

```
// Paso 1: Hacer login para obtener el token JWT
const credenciales = { username: "nuevo_usuario", password:
"ContraseñaSegura123" };
const respLogin = await fetch("http://localhost:8000/api/auth/login", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify(credenciales)
});
const datosLogin = await respLogin.json();
const tokenJWT = datosLogin.token;  // Suponiendo que la respuesta tiene un
campo 'token'
console.log("Token obtenido:", tokenJWT);
```

En este código, el cliente envía una petición POST a `/auth/login` con las credenciales en formato JSON (asegurándose de establecer `Content-Type: application/json`). La respuesta (`datosLogin`) contendrá el `token` JWT si las credenciales son válidas. Guardamos ese token (por ejemplo, en una variable, contexto de la app o *localStorage* en una aplicación web real) para usarlo en las siguientes llamadas.

1. **Usar el JWT para acceder a endpoints protegidos:** Una vez obtenido el token, el cliente lo enviará en el **header HTTP Authorization** de las peticiones posteriores. Por ejemplo, para obtener el perfil del usuario:

```
// Paso 2: Usar el token JWT en la cabecera Authorization para llamadas
autenticadas
const respPerfil = await fetch("http://localhost:8000/api/users/me", {
  method: "GET",
  headers: {
```

```
    "Authorization": `Bearer ${tokenJWT}`
  }
});
if (respPerfil.status === 200) {
  const perfilUsuario = await respPerfil.json();
  console.log("Datos del perfil:", perfilUsuario);
} else if (respPerfil.status === 401) {
  console.error("No autorizado: Token inválido o expirado");
}
```

Aquí hacemos una petición GET a `/users/me` incluyendo `Authorization: Bearer <tokenJWT>`. Si el token es válido y no expiró, la respuesta será 200 OK con los datos del usuario (perfil). Si el token es inválido o faltante, recibiremos un 401, tal como documentamos en nuestra especificación (y podríamos manejarlo mostrando un mensaje de error o redirigiendo al login).

El patrón es el mismo para cualquier endpoint protegido: siempre enviar el header `Authorization` con el token. Muchas librerías (como Axios) permiten configurar un *interceptor* para añadir automáticamente este header a cada petición después de login.

1. **Ejemplo usando Axios:** Con Axios, la sintaxis es similar:

```
import axios from "axios";
// ... (tras obtener tokenJWT mediante login como arriba)
axios.get("http://localhost:8000/api/users/me", {
  headers: { "Authorization": `Bearer ${tokenJWT}` }
})
.then(response => {
  console.log("Perfil usuario:", response.data);
})
.catch(error => {
  if (error.response && error.response.status === 401) {
    console.error("No autorizado o token expirado");
  }
});
```

Axios devuelve una promesa; en caso de éxito `response.data` tendrá el JSON del perfil. En caso de error 401, podemos capturarlo en el `.catch` y revisar el `status` para manejarlo.

En resumen, el flujo de consumo es: **Login -> Obtener Token -> Adjuntar Token en Authorization -> Llamar endpoints protegidos**. Gracias a la documentación OpenAPI, sabemos exactamente qué endpoints existen, qué datos esperan, qué respuestas devuelven, y que debemos enviar el JWT en el header *Authorization* para ciertos endpoints. Un desarrollador front-end puede consultar esta documentación (p. ej., via Swagger UI) y rápidamente entender cómo interactuar con la API.

Paso 9: Herramientas para validar y visualizar el YAML de OpenAPI

Escribir a mano un archivo OpenAPI en YAML puede ser propenso a errores de sintaxis o estructura. Afortunadamente, existen **herramientas** que facilitan la edición, validación y visualización de estas especificaciones:

- **Swagger Editor:** Una herramienta web (o instalable) que permite escribir la especificación OpenAPI en formato YAML/JSON y ver en tiempo real una vista previa de la documentación. Puedes usar la versión en línea en **editor.swagger.io**. Simplemente copia el YAML que hemos construido y pégalo allí; a la izquierda verás tu YAML con resaltado, y a la derecha la documentación renderizada. Swagger Editor destacará errores de sintaxis o de esquema en rojo, ayudándote a corregir el YAML. Es excelente para validar que tu archivo cumple con la especificación y para ver cómo se mostrará en Swagger UI.

- **Swagger UI:** Si quieres integrar la documentación interactiva en tu propio proyecto (por ejemplo, en tu aplicación Django), Swagger UI es un conjunto de archivos HTML/JS que lee tu archivo OpenAPI (YAML/JSON) y genera una página web interactiva donde puedes probar los endpoints. Podrías hostear tu archivo YAML/JSON en una URL (incluso en Django sirviéndolo como estático) y configurar Swagger UI para que lo cargue [9] [10]. Alternativamente, existen bibliotecas Django como **drf-yasg** o **drf-spectacular** que integran Swagger UI o ReDoc y pueden usar tu esquema manual. En nuestro caso, dado que documentamos manualmente, probablemente usaríamos la ruta estática + Swagger UI. Si prefieres no montar nada, la opción más rápida sigue siendo abrir Swagger Editor o Swagger UI online y pegar el YAML.

- **ReDoc:** Es otra herramienta popular para visualizar documentación OpenAPI, con un estilo distinto (más orientado a documentación estática). Puedes usarla similar a Swagger UI, cargando tu archivo YAML. ReDoc presenta la documentación en una sola página seccionada. Es útil para compartir documentación *legible* públicamente (mientras Swagger UI está más orientado a pruebas interactivas).

- **Validadores de OpenAPI:** Además de Swagger Editor (que valida automáticamente), existen utilidades como **OpenAPI Validator** o linters (por ejemplo, la CLI de Spectral o **openapi-cli** de Redocly) que pueden analizar tu YAML en busca de errores o malas prácticas. Si tu proyecto es grande, integrar una validación de esquema en tu CI/CD puede prevenir publicar documentación inconsistente.

- **Extensiones de IDE:** Si usas VSCode, hay extensiones como *Swagger Viewer* o *OpenAPI (Swagger) Editor* que te ayudan a editar archivos YAML de OpenAPI con autocompletado, validación y previsualización. Estas herramientas pueden hacer más cómoda la edición manual.

Para finalizar, recuerda que mantener la documentación actualizada es crucial. Cada vez que cambie tu API Django (nuevos endpoints, cambios en campos o respuestas), debes reflejarlo en el archivo OpenAPI. Una buena práctica es hacerlo parte del proceso de desarrollo: diseña primero o en paralelo el contrato (el YAML) y luego implementa/ajusta el código según este contrato. De esta forma, la documentación no se queda atrás de la implementación [11].

¡Enhorabuena! Has creado un tutorial completo con un archivo OpenAPI en YAML documentando endpoints de Django con autenticación JWT. Ahora puedes utilizar este archivo para generar documentación interactiva, o incluso para generar código cliente automáticamente si lo necesitas.

Referencias útiles:

- OpenAPI Specification y herramientas Swagger [2] [3]
- Mejores prácticas de estructura en OpenAPI (info, paths, components, security) [4] [6]
- Configuración de autenticación JWT en OpenAPI (Bearer Auth) [8]

¡Feliz documentación de APIs!

---

[1] [2] [3] Swagger vs OpenAPI: 4 diferencias principales ! — Wallarm
https://lab.wallarm.com/what/openapi-vs-swagger-una-aclaracion-en-profundidad/?lang=es

[4] [5] [6] [7] [11] Designing Python REST APIs with OpenAPI Specification Guide | MoldStud
https://moldstud.com/articles/p-designing-python-rest-apis-with-openapi-specification-a-comprehensive-guide

[8] Configure JWT Authentication for OpenAPI | Baeldung
https://www.baeldung.com/openapi-jwt-authentication

[9] [10] How to use custom YAML file as API documentation in Django REST Framework? - Stack Overflow
https://stackoverflow.com/questions/62296494/how-to-use-custom-yaml-file-as-api-documentation-in-django-rest-framework