Universidad La Salle

Seguridad Informática

Informe

Manual de usuario ApiGuardian

Luisfelipe Rodrigo Mamani Arosquipa

Decimo Semestre - Ingeniería de Software

2024

1. Introducción

En la actualidad, el desarrollo de aplicaciones modernas que exponen APIs públicas requiere garantizar seguridad, trazabilidad y cumplimiento normativo. Las APIs representan un punto crítico de acceso y deben ser protegidas contra ataques comunes como fuerza bruta, abuso de endpoints y exposición de datos sensibles.

API Guardian es una solución construida sobre Django que permite autenticar, autorizar y auditar el acceso a servicios web RESTful mediante buenas prácticas de seguridad, control de tokens y monitoreo de actividades. Se centra en el uso de JWT, rotación de tokens, rate-limiting y registro detallado de peticiones para proteger la integridad y disponibilidad de los servicios ofrecidos.

2. Objetivo del Proyecto

Desarrollar un sistema seguro de autenticación, autorización y auditoría para proteger APIs RESTful, aplicando medidas como control de acceso por tokens, limitación de peticiones, registros detallados de actividad, bloqueo por intentos fallidos y documentación interactiva para su uso adecuado.

3. Alcance

- Autenticación y registro de usuarios vía email y contraseña
- Gestión de tokens de acceso con JWT (rotación segura)
- Endpoints protegidos con control de acceso
- Middleware personalizado para auditoría de peticiones
- Documentación automática de la API (Swagger/OpenAPI 3)
- Rate-limiting por IP o usuario
- Bloqueo por intentos de acceso fallidos (fuerza bruta)
- Despliegue seguro en producción con HTTPS

4. Marco Teórico o Tecnológico

El desarrollo de apiGuardian se basa en un conjunto de tecnologías modernas del ecosistema Django, orientadas a construir una API segura, escalable y auditable. Este marco tecnológico permite implementar autenticación basada en JWT, control de acceso granular, auditoría detallada y documentación automatizada.

Seguridad y Autenticación

- **Django:** Framework web robusto en Python que sigue el patrón MTV (Modelo-Template-Vista). Se encarga del procesamiento de lógica, autenticación y ORM.

- **Django REST Framework (DRF):** Extiende Django para construir APIs RESTful. Permite serialización, vistas basadas en clases y autenticación de peticiones.

- **SimpleJWT:** Módulo para DRF que implementa autenticación mediante tokens JWT (JSON Web Token). Soporta tokens de acceso y refresh, rotación y blacklist.

- **dj-rest-auth:** Simplifica la autenticación REST. Provee endpoints para login, logout, registro, cambio de contraseña y verificación de email.

- **django-allauth:** Librería robusta para autenticación de usuarios. Soporta email/password y autenticación social (como Google, GitHub, etc.).

4.1 Estructura del Sistema

El sistema sigue una arquitectura modular, organizada en múltiples aplicaciones Django con responsabilidades separadas. Se adopta el patrón de diseño MVC (Modelo-Vista-Controlador), junto con prácticas RESTful para la estructura de los endpoints.

Módulos principales

Módulo	Descripción
api_guardian_auth	Contiene las vistas para login, logout, refresh de tokens, registro, etc.
api_guardian_users	Lógica de gestión de usuarios con roles personalizados (ej. managers).
api_guardian_auditoria	Middleware de auditoría y modelo AuditLog para registrar las acciones.

Auditoría y Protección

- django-axes: Protección contra ataques de fuerza bruta. Bloquea IPs o combinaciones IP+usuario tras varios intentos fallidos.

- Middleware de auditoría (AuditLogMiddleware): Middleware personalizado que registra información de cada petición (método, IP, usuario, cuerpo de la solicitud y respuesta, etc.).

Infraestructura y Herramientas

- **PostgreSQL:** Motor de base de datos robusto y escalable, usado para almacenar usuarios, tokens y logs.

- **Caddy v2:** Servidor web que actúa como proxy reverso con HTTPS automático usando certificados de Let's Encrypt.

- **Gunicorn:** Servidor WSGI que ejecuta la aplicación Django en producción.

- **drf-spectacular:** Genera documentación OpenAPI 3.0 (Swagger) automáticamente para los endpoints de la API.

5. Guía de Instalación del Proyecto (Manual de Usuario – Instalación)

Requisitos previos

- Python 3.11+

- PostgreSQL 13+

- pipenv o virtualenv

- Git

- Caddy (opcional en producción)

- Gunicorn (para WSGI)

- .env con las variables de entorno

Pasos para entorno local (Desarrollo)

- **Clonar el repositorio**

```
git clone https://github.com/lfuis201/apiGuardian.git

cd apiGuardian
```

- **Crear entorno virtual y activar**

```
python -m venv venv

source venv/bin/activate  # Linux/macOS

.\venv\Scripts\activate    # Windows
```

- **Instalar dependencias**

```
pip install -r requirements.txt
```

- **Configurar variables de entorno**

```
DB_NAME=nombre_bd

DB_USER=usuario

DB_PASSWORD=clave

DB_HOST=localhost

DB_PORT=5432

EMAIL_HOST_USER=tuemail@gmail.com

EMAIL_HOST_PASSWORD=claveemail
```

- **Aplicar migraciones**

```
python manage.py migrate
```

- **Crear superusuario**

```
python manage.py createsuperuser
```

- **Ejecutar el servidor**

```
python manage.py runserver
```

- **Acceder a Swagger UI**

http://127.0.0.1:8000/api/docs/

Despliegue en Producción (Gunicorn + Caddy)

- **Ejecutar Gunicorn**
  ```
  gunicorn api_guardian.wsgi:application --bind
  localhost:8000
  ```
- **Archivo Caddy**

  ```
  apiguardian.bashuabrand.com {

  reverse_proxy localhost:8000

  encode gzip

  tls your_email@example.com

  }
  ```

- **Iniciar Caddy**
  ```
  sudo caddy run --config Caddyfile
  ```

6. Requisitos del Sistema

6.1 Requisitos Funcionales

Los requisitos funcionales definen el conjunto de funcionalidades que el sistema **apiGuardian** debe ofrecer al usuario final o a otros sistemas consumidores. Estos incluyen el registro y autenticación segura de usuarios mediante tokens JWT, la protección contra ataques por fuerza bruta, la limitación de accesos por IP o usuario para prevenir abusos, y la auditoría detallada de todas las peticiones realizadas al sistema. Además, el sistema debe exponer documentación interactiva y actualizada de su API mediante Swagger, garantizando accesibilidad y trazabilidad completa para desarrolladores e integradores. Cada una de estas funciones contribuye a asegurar un entorno seguro, controlado y mantenible para el consumo de servicios RESTful.

ID	Nombre del Requisito	Descripción Breve
RF01	Autenticación y autorización con JWT	Permitir a los usuarios iniciar sesión y acceder con seguridad mediante tokens JWT.
RF02	Cifrado de tráfico	Garantizar que toda la comunicación cliente-servidor se realice bajo HTTPS.

RF03	Rate-limiting por usuario/IP	Limitar el número de peticiones por usuario o IP para evitar abuso del sistema.
RF04	Registro detallado de peticiones y respuestas	Registrar en base de datos las solicitudes y respuestas procesadas por la API.
RF05	Detección de uso indebido	Emitir alertas o bloquear accesos ante patrones sospechosos o intentos maliciosos.

6.1.1 Fichas Requisitos Funcionales

RF01 – Autenticación y autorización con JWT

Campo	Detalle
ID	RF01
Nombre	Autenticación y autorización con JWT
Descripción	Permitir a los usuarios autenticarse con email y contraseña, obteniendo tokens JWT para acceder a recursos protegidos.
Prioridad	Alta
Actor principal	Usuario final
Entrada	Email y contraseña
Salida esperada	Access token (60 min) y Refresh token (1 día)
Librerías usadas	dj-rest-auth, rest_framework_simplejwt

Ruta API	/api/login/

RF02 – Cifrado de tráfico entre cliente y API

Campo	Detalle
ID	RF02
Nombre	Cifrado de tráfico
Descripción	Asegurar que todas las peticiones se realicen mediante HTTPS para proteger la información en tránsito.
Prioridad	Alta
Actor principal	Usuario final
Requisito técnico	HTTPS obligatorio con certificados válidos
Herramienta usada	Caddy v2 (proxy inverso con TLS automático via Let’s Encrypt)
URL de prueba	https://apiguardian.bashuabrand.com/api

RF03 – Rate-limiting por usuario/IP

Campo	Detalle
ID	RF03
Nombre	Limitación de peticiones por usuario/IP

Descripción	Controlar el número de solicitudes permitidas por usuario o IP en un período determinado.
Prioridad	Alta
Actor principal	Usuario o atacante
Implementación sugerida	Uso de la librería django-ratelimit o middleware personalizado
Ejemplo de uso	Decorador @ratelimit(key='user', rate='10/m', method='POST', block=True)

RF04 – Registro detallado de peticiones y respuestas

Campo	Detalle
ID	RF04
Nombre	Registro de auditoría
Descripción	Registrar información como: usuario, IP, método, endpoint, status code y tiempo de respuesta.
Prioridad	Alta
Actor principal	Sistema
Herramienta usada	Middleware personalizado AuditLogMiddleware
Almacenamiento	Base de datos PostgreSQL (tabla AuditLog)

RF05 – Detección de uso indebido o comportamiento anómalo

Campo	Detalle
ID	RF05
Nombre	Detección de uso indebido
Descripción	Detectar patrones como múltiples intentos fallidos de acceso, uso inusual de endpoints o picos anómalos de tráfico.
Prioridad	Alta
Actor principal	Usuario malicioso
Herramienta usada	django-axes para login, y alertas personalizadas en middleware de auditoría
Resultado esperado	Bloqueo temporal + alerta en consola o email

6.2 Requisitos No Funcionales

Los requisitos no funcionales establecen las cualidades del sistema que aseguran su buen funcionamiento más allá de la lógica de negocio. En el caso de apiGuardian, se exige una arquitectura modular y escalable, que permita agregar nuevas funcionalidades sin afectar la estabilidad del sistema. Se requiere que el código sea mantenible y configurable fácilmente a través de variables de entorno (.env). El sistema debe entregar respuestas estandarizadas en formato JSON y mantener tiempos de respuesta óptimos incluso en producción. Asimismo, se enfatiza el uso de HTTPS para la transmisión segura de datos, y una gestión robusta de errores y registros para facilitar el monitoreo y mantenimiento del entorno.

1. Sistema escalable y modular RNF01.

2. Implementación rápida vía configuración .env RNF02.

3. Código limpio y mantenible RNF03.
4. Respuesta estandarizada en formato JSON RNF04.
5. Tiempo de respuesta aceptable en producción RNF05.

6.2.1 Fichas de Requisitos No Funcionales

RNF01 – Sistema escalable y modular

Campo	Detalle
ID	RNF01
Nombre	Escalabilidad y modularidad
Descripción	El sistema debe permitir la incorporación de nuevas funcionalidades sin alterar la arquitectura base.
Importancia	Alta
Evidencia de cumplimiento	Arquitectura basada en apps Django independientes y middlewares modulares.

RNF02 – Configuración rápida vía .env

Campo	Detalle
ID	RNF02
Nombre	Variables de entorno externas

Descripción	Los parámetros sensibles deben estar fuera del código fuente.
Importancia	Alta
Evidencia de cumplimiento	Uso de `python-decouple` para cargar `SECRET_KEY`, credenciales y conexión DB.

RNF03 – Código limpio y mantenible

Campo	Detalle
ID	RNF03
Nombre	Buenas prácticas de codificación
Descripción	El sistema debe tener un código legible, comentado y separado por responsabilidad.
Importancia	Alta
Evidencia de cumplimiento	Separación por apps, middlewares, configuraciones y vistas bien documentadas.

RNF04 – Respuesta JSON estandarizada

Campo	Detalle

ID	RNF04
Nombre	Formato de respuesta unificado
Descripción	Todas las respuestas de la API deben retornar objetos JSON.
Importancia	Media
Evidencia de cumplimiento	Respuestas DRF en JSON con `status`, `data` y `detail`.

RNF05 – Tiempo de respuesta aceptable

Campo	Detalle
ID	RNF05
Nombre	Rendimiento en producción
Descripción	Las peticiones deben responder en menos de 1 segundo bajo carga promedio.
Importancia	Media
Evidencia de cumplimiento	Pruebas manuales con Postman y Swagger muestran tiempos medios de 200–400 ms.

7. Modelado del Sistema (Diagramas)

El modelado del sistema permite representar visualmente la arquitectura, estructura de clases, flujo de procesos y relaciones de datos del proyecto **apiGuardian**. A través de distintos tipos

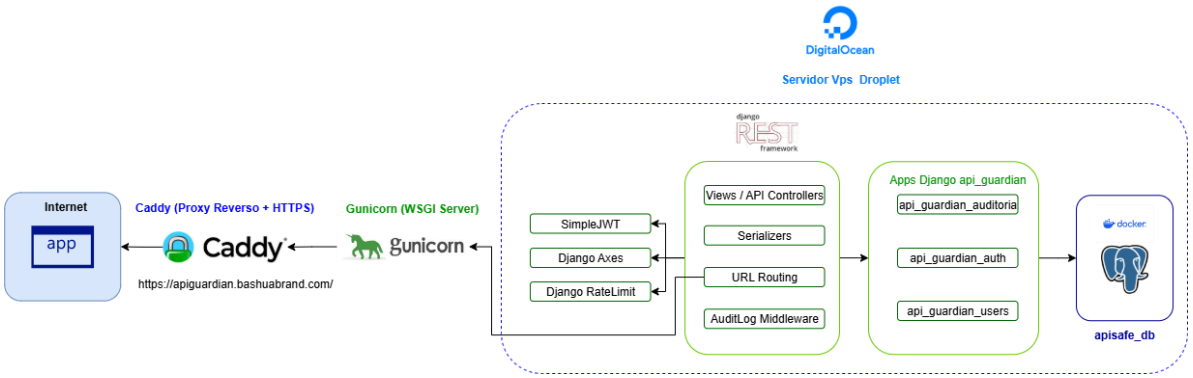
de diagramas, se facilita la comprensión de cómo interactúan los componentes del sistema, tanto a nivel lógico como físico.

7.1 Diagrama de Arquitectura General

Descripción
Este diagrama presenta la visión global de los componentes principales que conforman el entorno de producción del sistema. Muestra cómo los usuarios interactúan con el sistema a través del navegador o clientes externos, cómo las peticiones pasan por el servidor proxy Caddy (que gestiona HTTPS y redirige al backend), cómo Gunicorn sirve la aplicación Django, y cómo esta se conecta con la base de datos PostgreSQL para almacenar o consultar información.

Objetivo
Comprender la infraestructura desplegada en producción y el flujo general de peticiones.

Diagrama



7.2 Diagrama de Clases (Modelo de Django)

7.2.1 Diagrama de Clases – Módulo de Usuarios y Autenticación

Descripción General

Este diagrama representa las clases clave relacionadas con la gestión de usuarios, autenticación y autorización dentro del sistema apiGuardian, el cual se apoya en la arquitectura nativa de Django y extensiones como dj-rest-auth, django-allauth y djangorestframework-simplejwt.

Se muestra la clase base User (proporcionada por django.contrib.auth.models.User) junto con sus relaciones con otras entidades como:

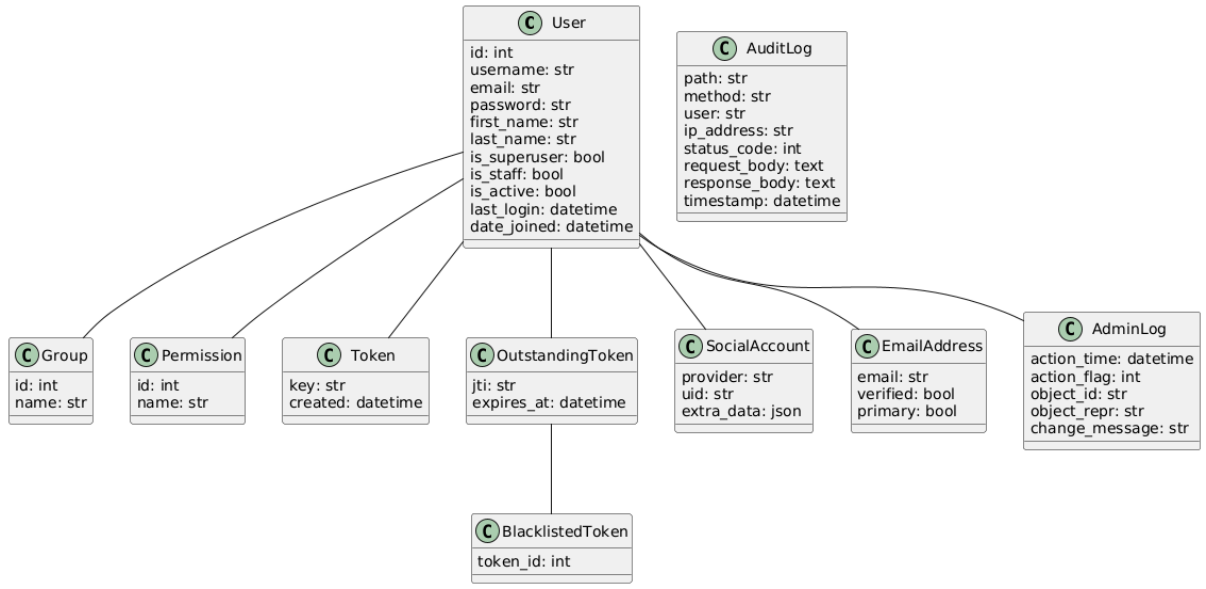
- **Grupos (Group):** permite asignar permisos colectivos.
- **Permisos (Permission):** asignaciones individuales de capacidades específicas.

- **Token (authtoken_token):** tokens para autenticación clásica.
- **OutstandingToken y BlacklistedToken:** para el manejo de tokens JWT (autenticación moderna).
- **SocialAccount y EmailAddress:** extensiones provistas por django-allauth para login social y gestión de múltiples correos.
- **AuditLog:** clase personalizada para registrar auditoría de peticiones.
- **AdminLog:** clase de auditoría automática de acciones administrativas de Django.

Objetivo

El objetivo de este diagrama es visualizar la estructura de clases y relaciones del módulo de autenticación y gestión de usuarios dentro del sistema apiGuardian. Permite comprender cómo se modelan los usuarios, cómo interactúan con componentes de seguridad como JWT, grupos y permisos, y cómo se extienden estas funcionalidades mediante autenticación social y registros de auditoría.

Diagrama



7.3 Diagrama Entidad-Relación (ER)

Este diagrama representa gráficamente las tablas de la base de datos generadas automáticamente por los modelos del sistema Django (User, AuditLog, Group, Token, SocialAccount, entre otros). A diferencia del diagrama de clases, el enfoque aquí es físico, es decir, en cómo se almacenan y relacionan los datos en la base de datos PostgreSQL.

Incluye:

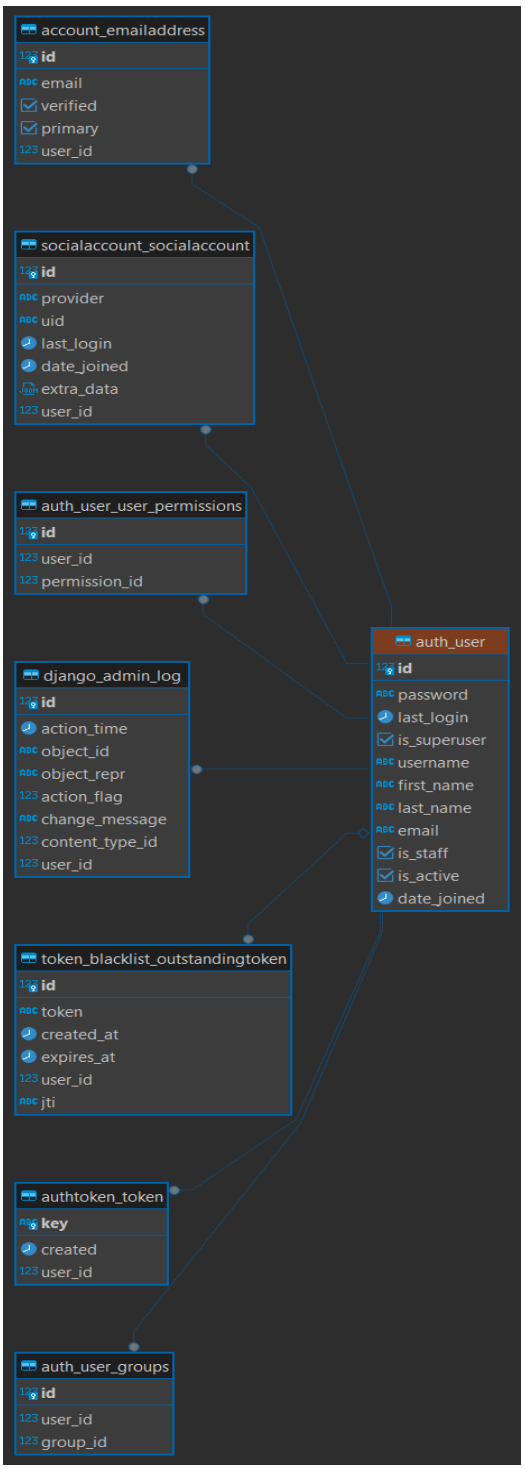
- **Claves primarias (PK) y foráneas (FK)**
- **Relaciones uno a uno**, **uno a muchos** y **muchos a muchos**
- **Campos relevantes** como user_id, email, token, group_id, timestamp, etc.
- Tablas de terceros como token_blacklist, auth_user_groups, account_emailaddress, socialaccount_socialaccount

Objetivo

El objetivo de este diagrama es **visualizar la estructura lógica y física** de la base de datos del sistema apiGuardian, permitiendo:

- Entender cómo se almacenan las entidades clave como usuarios, tokens y registros de auditoría.
- Verificar la **integridad referencial** mediante claves foráneas entre entidades relacionadas.
- Detectar posibles **problemas de diseño o redundancia de datos**.
- Servir como **base para futuras extensiones** o integraciones externas que requieran acceso directo a la base de datos.

Diagrama



7.4 Diagrama de Flujo o de Actividades

7.4.1 Flujo 1: Autenticación con Bloqueo (Axes) + Registro de Auditoría (AuditLogMiddleware)

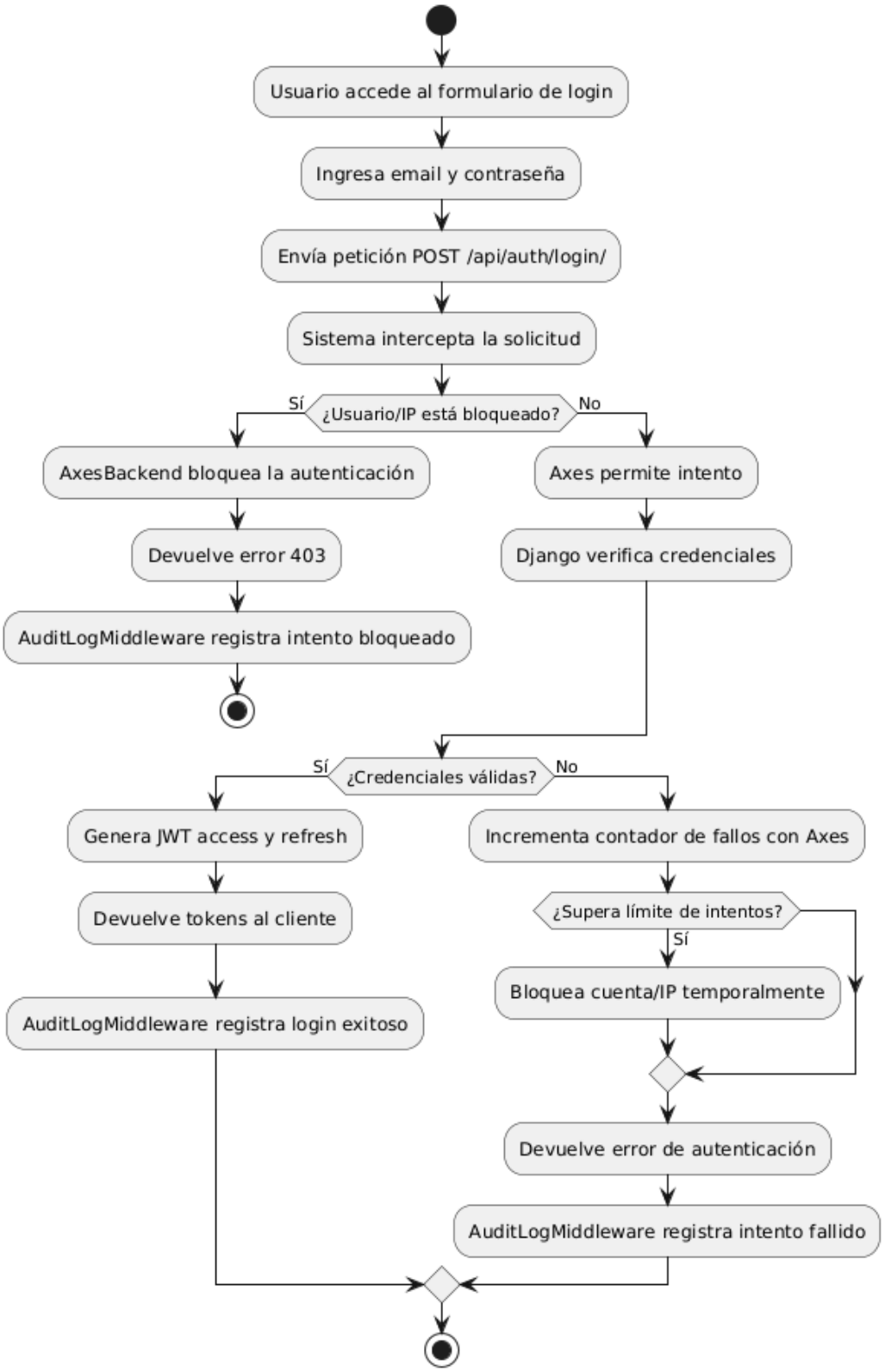
Descripción

Este flujo describe el proceso completo de autenticación de un usuario, integrando control de seguridad por fuerza bruta con django-axes y registro de todas las peticiones mediante el middleware AuditLogMiddleware. Antes de validar las credenciales, el sistema verifica si el usuario/IP está bloqueado. Todas las respuestas son auditadas, independientemente del resultado.

Objetivo

Fortalecer la autenticación de usuarios con bloqueo automático por intentos fallidos y trazabilidad completa de las acciones mediante logs persistentes.

Diagrama de Actividades



7.4.2 Flujo 2: Gestión de Managers (Crear y Listar)

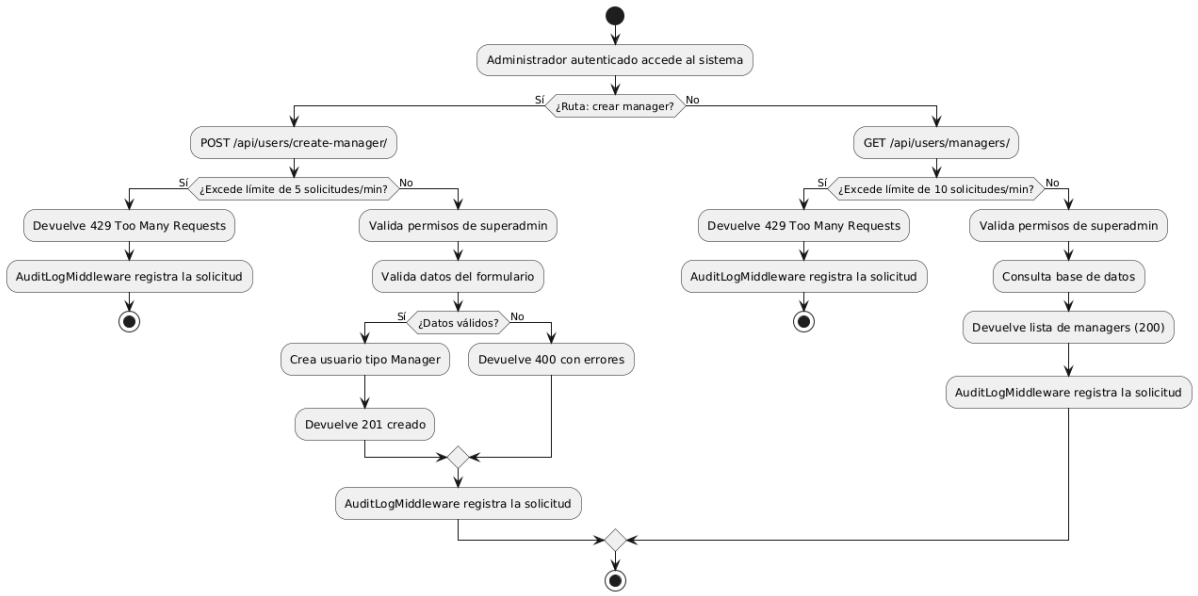
Descripción

Este flujo representa el proceso administrativo donde un usuario con permisos de administrador puede crear nuevos usuarios tipo "Manager" y consultar la lista de managers registrados. El acceso está restringido mediante autenticación JWT.

Objetivo

Permitir la delegación controlada de responsabilidades administrativas mediante la creación y gestión de usuarios de tipo Manager, respetando las políticas de acceso.

Diagrama de Actividades



8. Documentación de API

Esta sección presenta un resumen de los principales endpoints de la API, mostrando los payloads de solicitud y las respuestas más relevantes, incluyendo manejo de errores, autenticación JWT y mecanismos de seguridad como limitación de tasa (rate limit) y bloqueo por fuerza bruta (Axes). Para una documentación completa, con detalles adicionales, ejemplos y pruebas interactivas, consulte el panel Swagger UI.

8.1 LOGIN

Endpoint: POST /api/auth/login/
Requiere: Credenciales válidas (email y password)

Payload (Request Body)

```
{

  "email": "admin@mail.com",
```

```
  "password": "12345678"
}
```


Response 200 – Login Exitoso


```
{
  "access": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "refresh": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
}
```


Response 401 – Credenciales inválidas


```
{
  "non_field_errors": [
    "Unable to log in with provided credentials."
  ]
}
```


Response 403 – Bloqueo por Fuerza Bruta (Axes)


```
{
  "detail": "Demasiados intentos fallidos. Tu cuenta o IP ha sido bloqueada temporalmente."
}
```


8.2 REFRESH TOKEN

Endpoint: POST /api/auth/token/refresh/
Requiere: Token de refresh válido

Payload


```
{
  "refresh": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
}
```

Response 200 – Token renovado


```
{
  "access": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUz..."
}
```


Response 401 – Token expirado o inválido


```
{
  "code": "token_not_valid",
  "detail": "Token is invalid or expired",
  "messages": [
    {
      "token_class": "RefreshToken",
      "token_type": "refresh",
      "message": "Token has expired"
    }
  ]
}
```


8.3 CREAR MANAGER

Endpoint: POST /api/users/create-manager/
Requiere: Rol SuperAdmin
Rate Limit: 5 solicitudes por minuto

Payload


```
{
  "username": "manager001",
  "email": "manager001@mail.com",
  "password": "securePassword123"
```

}

Response 201 – Manager creado

{

"detail": "Manager creado correctamente"

}

Response 400 – Validación fallida

{

"email": ["Este campo debe ser único."],

"password": ["La contraseña debe tener al menos 8 caracteres."]

}

Response 403 – No autorizado

{

"detail": "No tienes permisos para realizar esta acción."

}

Response 429 – Límite de peticiones alcanzado

{

"detail": "Límite excedido para esta ruta. Espera un momento antes de volver a intentarlo."

}

8.4 LISTAR MANAGERS

Endpoint: GET /api/users/managers/
Requiere: Rol SuperAdmin
Rate Limit: 10 solicitudes por minuto

Response 200 – Lista de managers

[

{

```
    "id": 4,
    "username": "manager001",
    "email": "manager001@mail.com",
    "date_joined": "2025-07-03T14:30:00Z"
  },
  {
    "id": 5,
    "username": "manager002",
    "email": "manager002@mail.com",
    "date_joined": "2025-07-03T15:20:00Z"
  }
]
```


Response 403 – Sin permisos


```
{
  "detail": "No tienes permisos para acceder a esta vista."
}
```


Response 429 – Rate Limit alcanzado


```
{
  "detail": "Límite excedido para esta ruta. Espera un momento antes de volver a intentarlo."
}
```


JSON de Auditoría (AuditLog)

El modelo AuditLog registra información detallada de cada petición y respuesta que pasa por la API para fines de trazabilidad y seguridad.

```
from django.db import models

from django.utils.timezone import now
```

```python
class AuditLog(models.Model):
    path = models.CharField(max_length=255)
    method = models.CharField(max_length=10)
    user = models.CharField(max_length=255, null=True, blank=True)
    ip_address = models.GenericIPAddressField(null=True, blank=True)
    status_code = models.IntegerField()
    request_body = models.TextField(null=True, blank=True)
    response_body = models.TextField(null=True, blank=True)
    timestamp = models.DateTimeField(default=now)

    class Meta:
        ordering = ['-timestamp']

    def __str__(self):
        return f"[{self.timestamp}] {self.method} {self.path} ({self.status_code})"
```

Campo	Tipo	Descripción
path	Cadena de texto	Ruta o endpoint al que se hizo la petición (por ejemplo, /api/auth/login/).

method	Cadena de texto	Método HTTP usado en la petición (GET, POST, PUT, etc.).
user	Cadena de texto	Nombre de usuario autenticado que realizó la petición. Puede ser null si es anónimo.
ip_address	Dirección IP	Dirección IP del cliente que realizó la petición. Puede ser null si no disponible.
status_code	Entero	Código HTTP de respuesta devuelto (por ejemplo, 200, 401, 429).
request_body	Texto	Contenido del cuerpo de la petición (payload enviado). Puede estar vacío si no aplica.
response_body	Texto	Contenido de la respuesta enviada al cliente. Puede estar vacío si no aplica.
timestamp	Fecha y hora	Marca temporal que indica cuándo se registró la petición y respuesta.

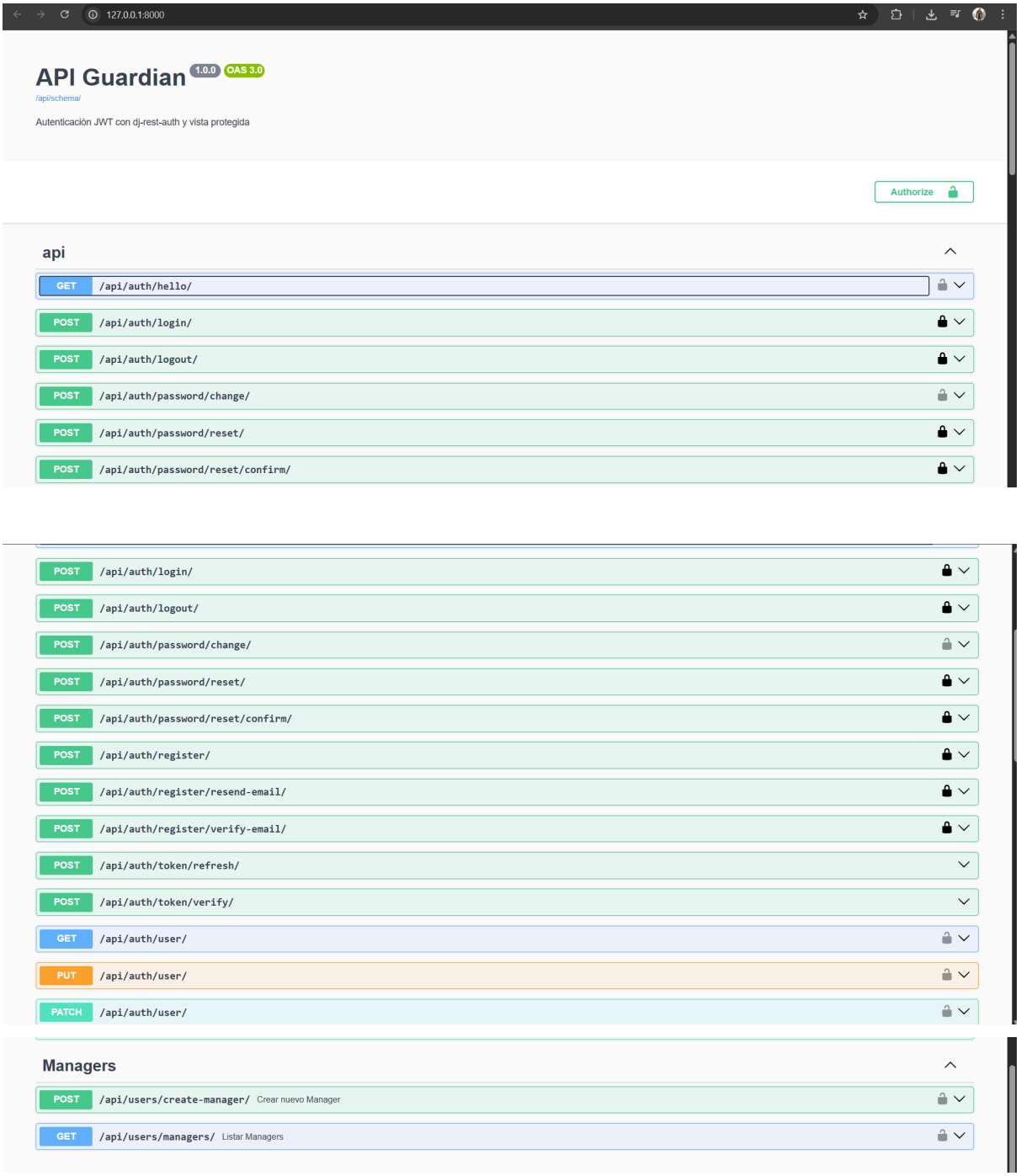
Este registro es fundamental para auditar el uso de la API, detectar patrones anómalos o intentos de acceso indebido, y cumplir con normativas de seguridad y trazabilidad.

8.5 Resumen de Errores Especiales

Código	Causa	JSON de respuesta
401	Token inválido o no	{ "detail": "Authentication credentials were not provided." }

	autenticado	
403	Usuario sin permisos suficientes	{ "detail": "No tienes permisos para realizar esta acción." }
429	Exceso de peticiones (Rate Limit)	{ "detail": "Límite excedido para esta ruta. Espera un momento antes de volver a intentarlo." }
403	Bloqueo por fuerza bruta (Axes)	{ "detail": "Demasiados intentos fallidos. Tu cuenta o IP ha sido bloqueada temporalmente." }

8.6 Swagger de ApiGuardian



9. Conclusiones del Manual de Usuario

El presente manual ha detallado paso a paso la instalación, configuración y uso del sistema **API Guardian**, una solución integral para proteger APIs REST con altos estándares de seguridad, trazabilidad y control de acceso.

A lo largo de la guía, el usuario ha podido:

- Comprender cómo funciona el sistema de autenticación mediante JWT y su integración con `dj-rest-auth` y `SimpleJWT`.

- Registrar nuevos usuarios, verificar correos electrónicos y gestionar sesiones de forma segura.

- Visualizar el registro completo de cada interacción con la API gracias al sistema de **auditoría automática**.

- Experimentar el funcionamiento del **rate limiting** y el bloqueo por fuerza bruta con la ayuda de `django-axes`.

- Navegar por la documentación completa e interactiva generada automáticamente mediante Swagger.

- Acceder y gestionar roles específicos, como los usuarios **Manager**, con control granular de permisos.

El sistema está diseñado para ser utilizado por desarrolladores backend, equipos DevSecOps, testers y personal técnico que busque una solución confiable y extensible para asegurar el acceso a sus APIs.

10. Recomendaciones y Mejoras Futuras

Aunque el sistema API Guardian es funcional y robusto, se sugiere tener en cuenta las siguientes recomendaciones para extender sus capacidades en futuras versiones:

- **Habilitar autenticación en dos pasos (2FA):** Reforzar la seguridad de acceso implementando autenticación por correo electrónico o aplicaciones móviles.

- **Integrar dashboards administrativos:** Incorporar paneles visuales que muestren estadísticas de accesos, bloqueos, intentos fallidos y auditoría en tiempo real.

- **Exportar los logs a sistemas externos:** Permitir el análisis de registros a través de herramientas como **Elasticsearch** y **Kibana** para análisis forense o monitoreo avanzado.

- **Mejorar la cobertura de pruebas automatizadas:** Añadir pruebas unitarias y de integración para garantizar la calidad del sistema, utilizando herramientas como `pytest`.

- **Preparar despliegues con contenedores:** Facilitar la instalación en producción mediante Docker y orquestadores como Docker Compose o Kubernetes.

- **Agregar notificaciones automáticas:** Enviar alertas vía correo o webhook cuando se detecten comportamientos inusuales (como múltiples bloqueos o picos de tráfico).