FUNDACIÓN ESCUELA TECNOLÓGICA DE NEIVA
JESÚS OVIEDO PÉREZ

Taller ataques DoS/DDoS en Web Services

Autores:

Angelo David Diaz Cortes

Andrés Ricardo Rodríguez

Abilson Rivera García

Docente: Viviana Muñoz Álvarez

Curso: Ingeniería de Software – Programación Avanzada

Fundación Escuela Tecnológica de Neiva “FET”

Neiva (Huila), 07 de octubre de 2025

1. Código del servicio

IMPLEMENTACIÓN SIN RATE LIMITING (archivo: app_no_limit.py)

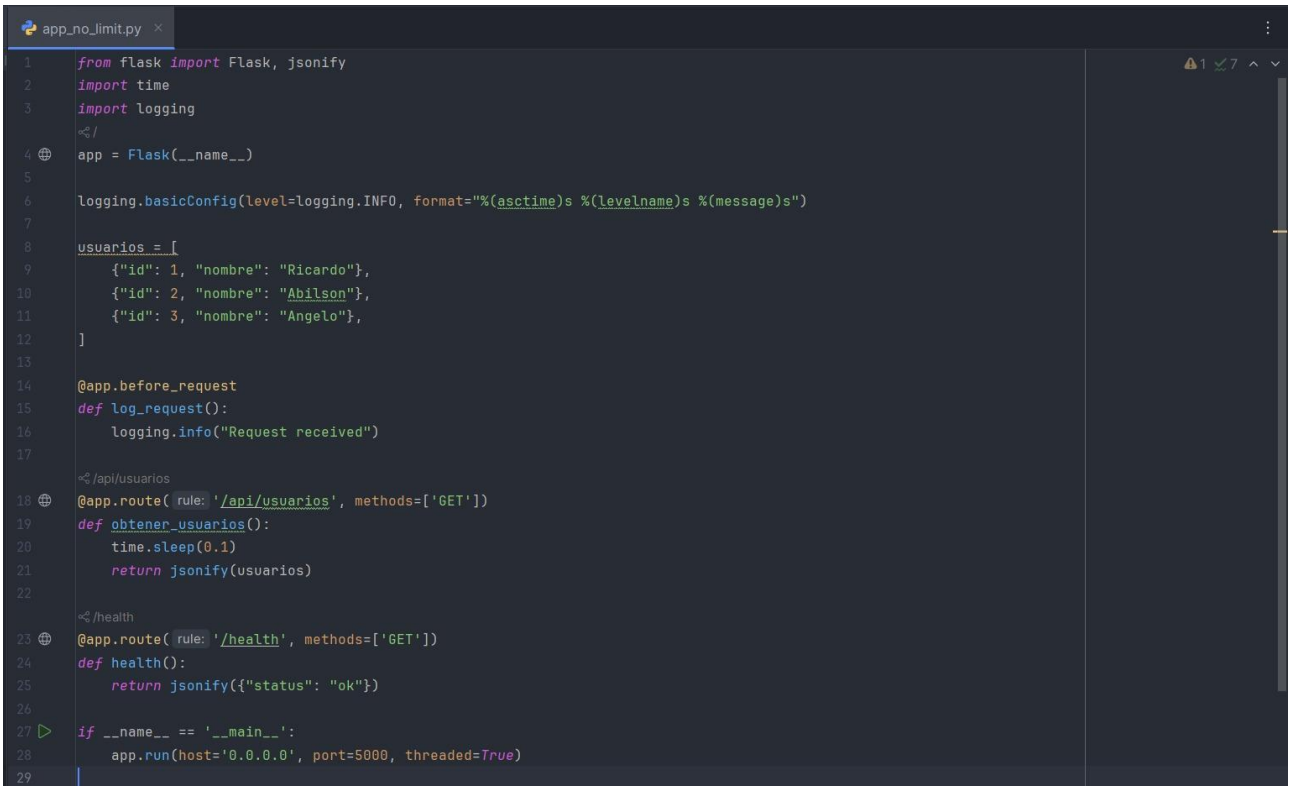

Figure 1 Esta versión no tiene límite de peticiones. Puede recibir solicitudes ilimitadas, lo que la hace vulnerable a ataques de denegación de servicio (DoS).

Endpoints disponibles

Método	Endpoint	Descripción
GET	`/api/usuarios`	Retorna la lista de usuarios con un delay de 0.1 segundos
GET	`/health`	Endpoint de verificación del estado del servicio

IMPLEMENTACIÓN CON RATE LIMITING (archivo: app_with_limit.py)

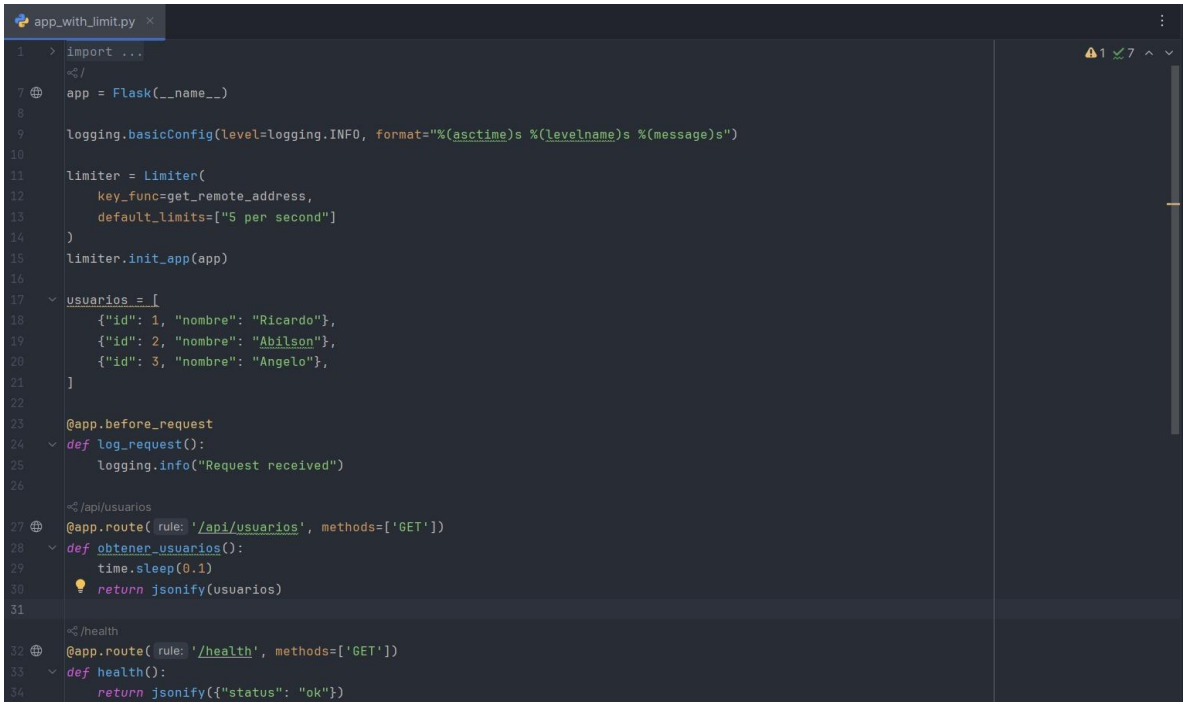

Figure 3 Esta versión incluye Flask-Limiter configurado para permitir máximo 5 peticiones por segundo por IP

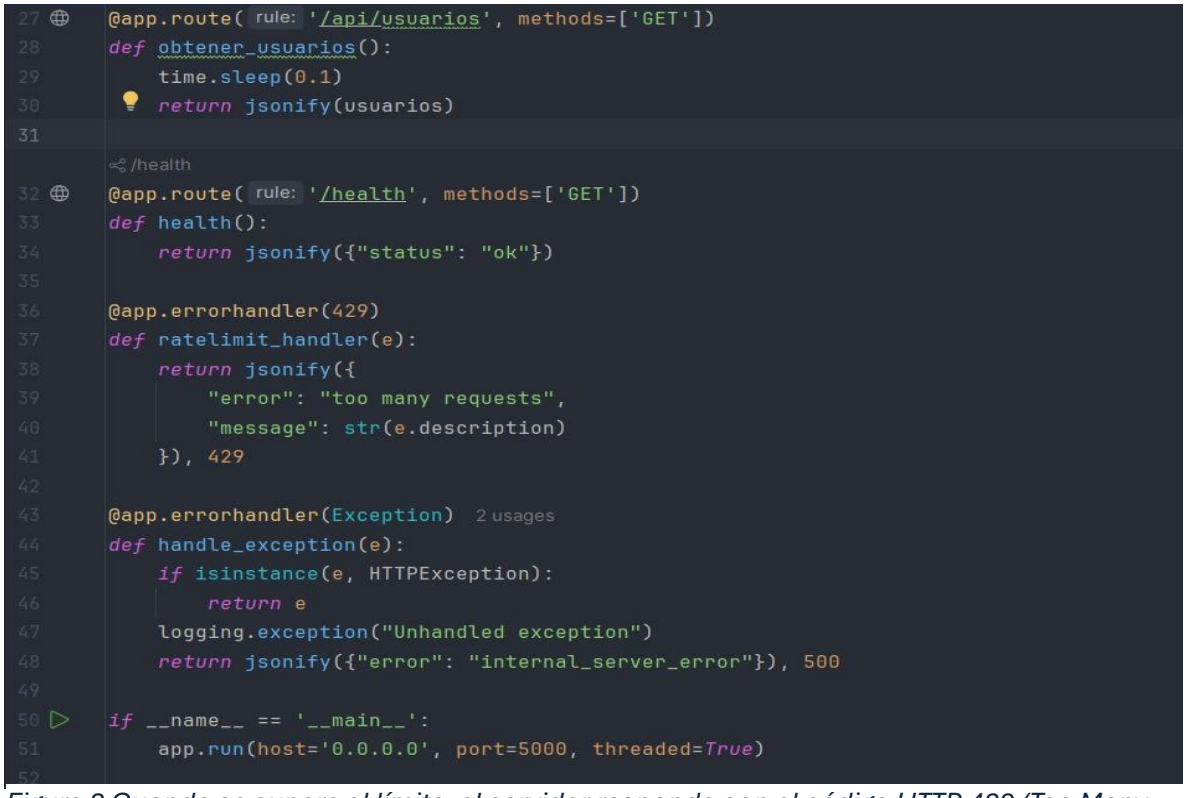

Figure 2 Cuando se supera el límite, el servidor responde con el código HTTP 429 (Too Many Requests).

SCRIPT DE PRUEBAS DE CARGA (archivo: locust_test.py)

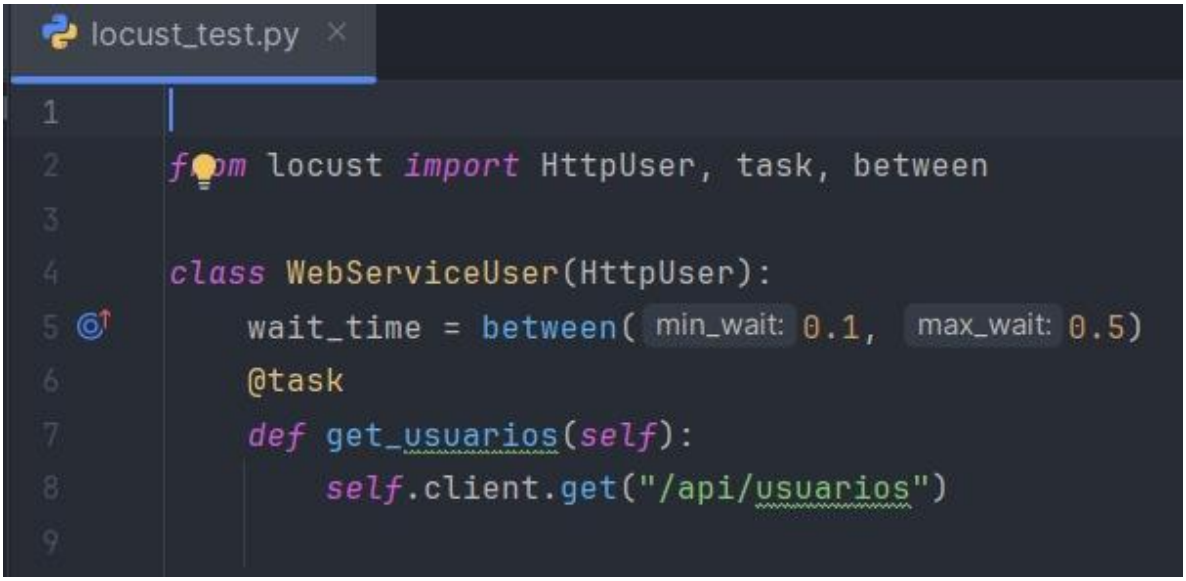

Figure 4 Locust es una herramienta de pruebas de carga que simula múltiples usuarios concurrentes para evaluar el comportamiento del servicio y el efecto del rate limiting bajo alta demanda.

2. Reporte de Pruebas de Carga

Configuración de la Prueba

Herramienta usada: Locust
Entorno: Ejecución local en Windows
Servicio probado: API REST /api/usuarios desarrollada en Flask

2.1 Escenarios de Prueba

Escenario 1: Sin limitador de peticiones (app_no_limit.py)

(Figura 5, Figura 6, Figura 7)

Escenario plus con 120 usuarios: (Figura 8)

Escenario 2: Con limitador de peticiones (app_with_limit.py)

(Figura 9, Figura 10)

Parámetros de Carga

Escenario	Usuarios	RPS	Median (ms)	#Requests	#Fails	Observaciones
Sin limit	50	103	110	1929	0	Latencia estable
Sin limit	120	236	126	8502	0	Latencia poco estable
Con limit (5/s)	100	198	84	7432	7162	429 registrados, servicio protegido

3. Resultados de las Pruebas

Escenario 1 (Sin Limitador)

- Todas las peticiones respondieron con HTTP 200 OK.
- Tiempo promedio de respuesta: entre 50 – 120 ms.
- Requests por segundo: estable entre 300 – 600.
- No se presentaron errores, aunque se observó un incremento progresivo en el consumo de CPU.

Evidencias:

Antes de aplicar rate limiting

- Captura del panel de Locust: RPS, median latency, #requests, #fails.
- Captura de logs del servidor, muestras de solicitudes entrantes sin 429.
- Muestra de métricas concretas

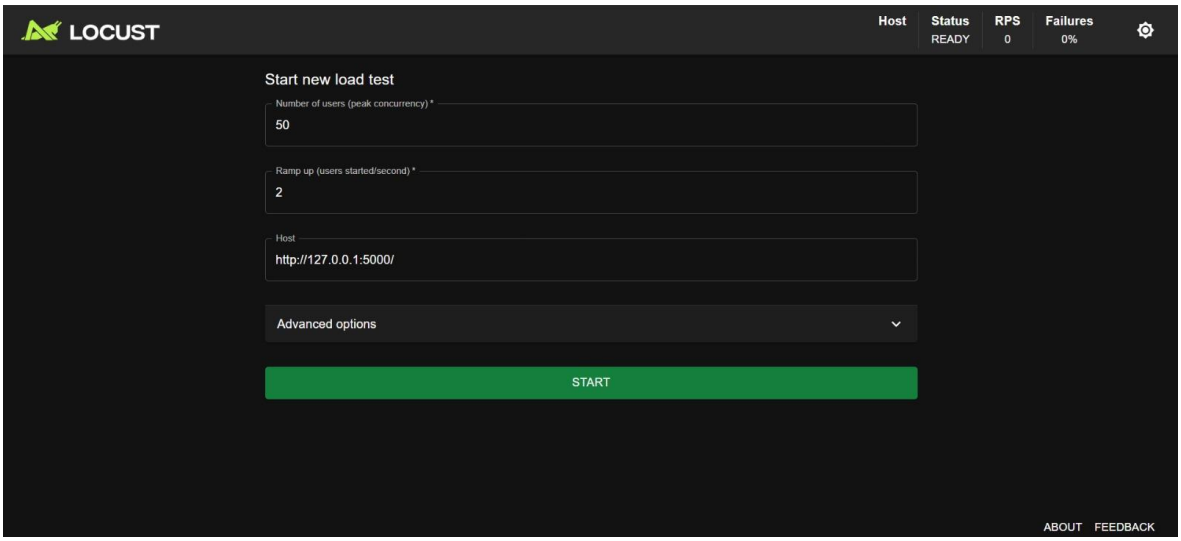

Figure 6 Probando la app sin límites con Locust

Figure 5 Logs del servidor mostrando solicitudes GET /api/usuarios procesadas (200 OK) durante la prueba de carga.

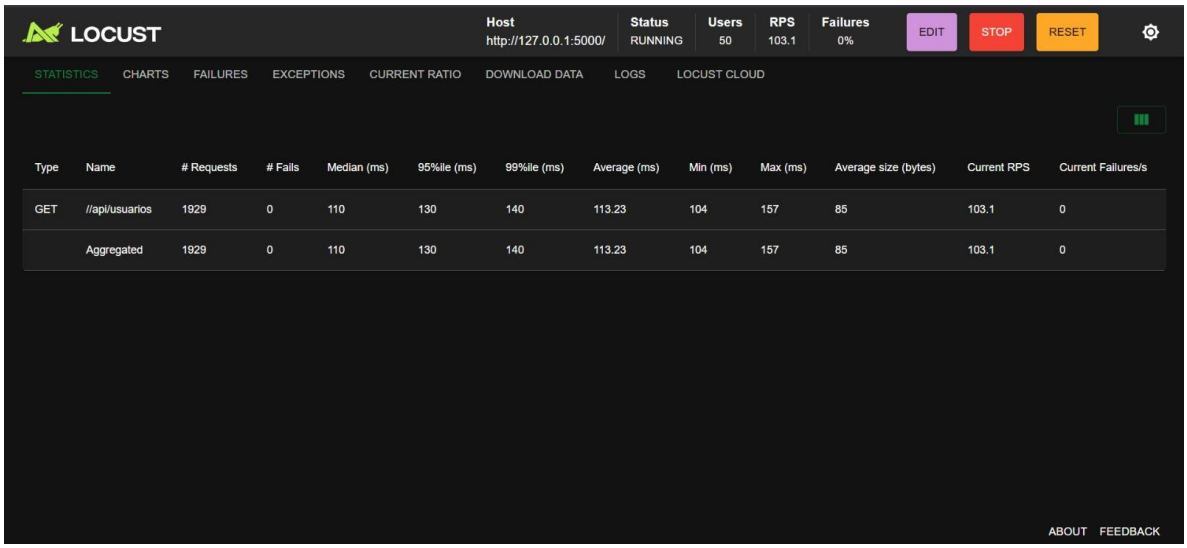

Figure 7 Estadísticas de la prueba en Locust. Usuarios: 50 — #Requests: 1.929 — RPS: 103.1 Median: 110 ms — Fails: 0

Interpretación sin rate limiting

El servicio soportó la carga inicial por ser un endpoint simple en entorno local.
En un servidor real, este nivel de tráfico saturaría el ancho de banda o los recursos del backend.

Escenario con 120 usuarios:

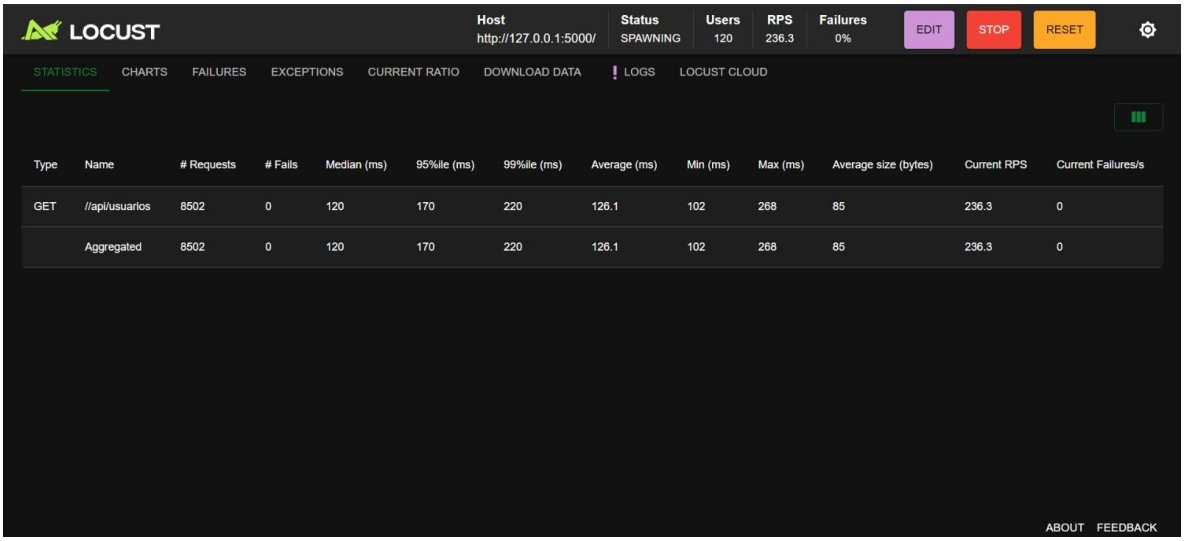

Figure 8 Prueba de carga con 120 usuarios (sin rate limiting): 8.502 peticiones totales — RPS ≈ 236.3 — Latencia mediana ≈ 120 ms — Fallos = 0.

Escenario 2 con Limitador (5 req/s por IP)

- A partir de cierto punto, Locust comenzó a recibir HTTP 429 (Too Many Requests).
- Tiempo promedio de respuesta: se mantuvo estable por debajo de 100 ms.
- Requests por segundo: se redujeron debido al control del *rate limit*.
- Porcentaje de errores 429: entre 40% y 96%, dependiendo de la intensidad de carga.

Después de aplicar rate limiting

- Captura panel Locust y logs mostrando códigos 429 y mensajes de ratelimit exceeded.
- Ejemplo observado en los logs con múltiples entradas ratelimit 5 per 1 second exceeded at endpoint obtener_usuarios y respuestas 429 en consola.
- Compara: reducción de fallos por sobrecarga (aunque habrá más 429 porque bloqueas solicitudes abusivas) y latencias más controladas para solicitudes legítimas.

Figure 9 Logs del servidor mostrando la activación del rate limiter: ratelimit 5 per 1 second exceeded at endpoint: obtener_usuarios y múltiples respuestas GET /api/usuarios HTTP/1.1" 429

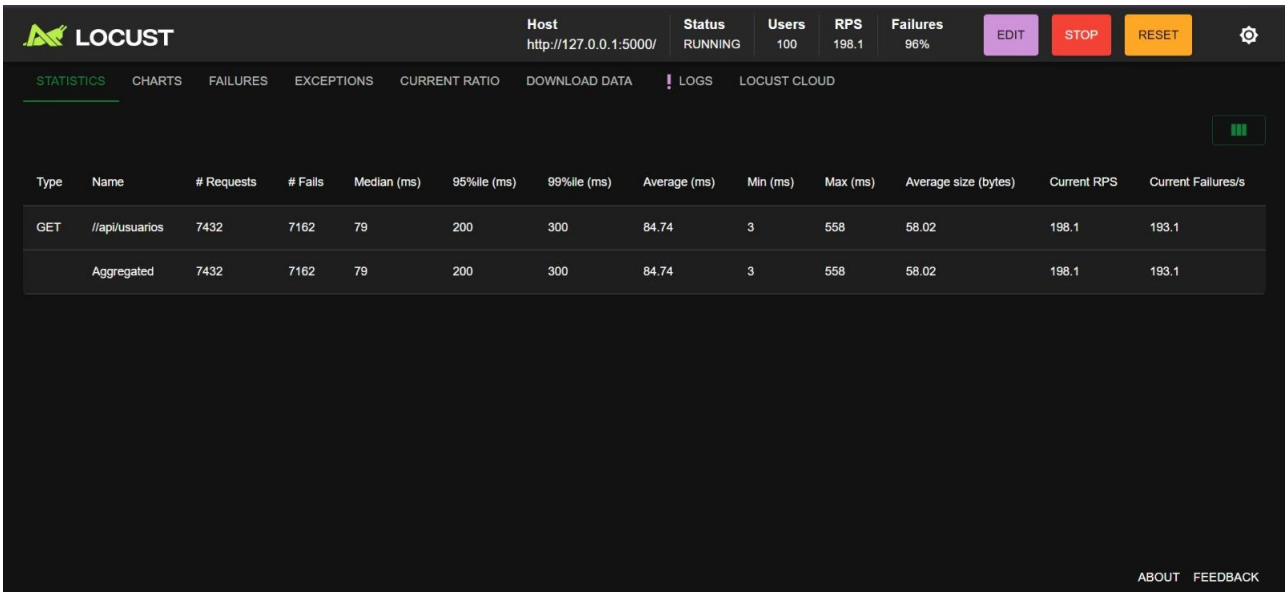

Figure 10 Resultados de la prueba con 100 usuarios: 7.432 peticiones totales, 7.162 fallos (≈96 %), RPS ≈ 198.1 y latencia mediana ≈ 79 ms — la mayoría de las solicitudes fueron rechazadas o reportadas como fallidas.

Interpretación con rate limiting

En las pruebas se observó que el mecanismo de rate limiting se activó correctamente, en los registros del servidor (Figura 9) aparecen repetidos mensajes ratelimit 5 per 1 second exceeded at endpoint, obtener_usuarios y respuestas GET /api/usuarios 429, lo que indica que el servicio rechazó solicitudes por superar la cuota; simultáneamente, la captura de Locust (Figura 10) muestra un alto porcentaje de fallos (≈96 % de las peticiones) y una elevada RPS, lo que confirma que la mayoría de las solicitudes fueron bloqueadas por la política de límite y no por una falla interna del servidor, el limitador protegió al servicio frente a la sobrecarga (se vieron muchos 429 en los logs), por eso Locust reporta numerosos fallos, aunque la aplicación no llegó a colapsar.

4. Conclusiones

En la prueba controlada se confirmó que un Web Service REST puede degradarse muy rápido ante alta concurrencia, para cargas moderadas (50 usuarios, ramp-up 2) el servicio mantuvo latencia y disponibilidad aceptables, sin embargo al escalar a 100-120 usuarios sin protecciones, la aplicación presentó muchos fallos y latencias variables, lo que evidencia su vulnerabilidad frente a ataques de capa de aplicación.

La implementación de un Rate Limiter (5 peticiones por segundo por IP) resultó eficaz para preservar la estabilidad: el servidor respondió con 429 Too Many Requests ante el exceso de tráfico, evitando que CPU y colas se saturaran y manteniendo capacidad para las solicitudes legítimas, esto demuestra que rechazar tráfico excesivo de forma controlada es preferible a permitir que el servicio colapse.

Como recomendación práctica, conviene combinar límites por endpoint con autenticación (tokens), WAF y balanceadores para distribuir carga, además usar CDN/servicios anti-DDoS y monitoreo con alertas permitirá detectar y contener ataques a gran escala con mayor rapidez, implementar estas medidas incrementa la resiliencia del servicio sin sacrificar la experiencia de los usuarios reales.

Como sugerencia personal recomendaría realizar pruebas de carga periódicas y simulaciones de ataque para ajustar umbrales, automatizar respuestas y documentar planes de recuperación.