UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Infraestructura de *backend* para la gestión de datos y la creación de formularios adaptables en el sector agrícola

Trabajo de graduación en modalidad de trabajo profesional presentado por
Linda Inés Jiménez Vides
para optar al grado académico de Licenciada en Ingeniería en Ciencias de la Computación y Tecnologías de la Información

Guatemala,

2025

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

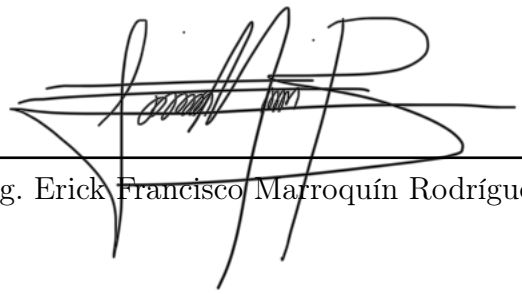
Infraestructura de *backend* para la gestión de datos y la creación de formularios adaptables en el sector agrícola

Trabajo de graduación en modalidad de trabajo profesional presentado por
Linda Inés Jiménez Vides
para optar al grado académico de Licenciada en Ingeniería en Ciencias de la Computación y Tecnologías de la Información

Guatemala,

2025

Vo.Bo.:

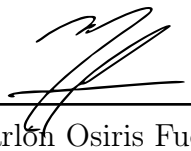
(f) 

Ing. Erick Francisco Marroquín Rodríguez

Tribunal Examinador:

(f) 

Ing. Erick Francisco Marroquín Rodríguez

(f) 

Ing. Marlón Osiris Fuentes López

Fecha de aprobación: Guatemala, 22 de noviembre de 2025.

Prefacio

El presente trabajo de graduación representa el cierre de una etapa académica y, al mismo tiempo, el inicio de una nueva etapa profesional. A lo largo del desarrollo de este proyecto, no solo se aplicaron conocimientos técnicos adquiridos durante la carrera, sino que también se enfrentaron retos reales que demandaron creatividad, disciplina y adaptabilidad.

La motivación detrás de este trabajo surge de la necesidad concreta de una institución agrícola por contar con herramientas tecnológicas propias y eficientes para la gestión de datos en campo. Esta necesidad no solo ofreció un contexto práctico valioso, sino que también evidenció cómo la ingeniería en ciencias de la computación puede contribuir directamente a resolver problemáticas del entorno.

Este documento es el resultado de meses de investigación, diseño, pruebas y validaciones. Sin embargo, más allá de la parte técnica, este proyecto también refleja el aprendizaje humano: la importancia del trabajo en equipo, la comunicación con usuarios reales y la responsabilidad de construir soluciones que tengan un impacto tangible.

Agradezco profundamente a todas las personas que brindaron su apoyo y acompañamiento durante este proceso: familiares, docentes y especialmente a mis compañeros de grupo, cuyo compromiso y colaboración fue fundamental para la integración completa de la herramienta. Finalmente, agradezco al equipo del Ingenio Santa Ana que confió en esta propuesta. Su orientación y confianza fueron fundamentales para concretar este trabajo.

Índice

Prefacio III

Lista de figuras VII

Lista de cuadros VIII

Lista de abreviaturas y siglas IX

Resumen X

Abstract XI

1. Introducción 1

2. Antecedentes 2

3. Justificación 3

4. Objetivos 4
4.1. Objetivo general . . . 4
4.2. Objetivos específicos . . . 4

5. Marco teórico 5
A. *Framework* . . . 5
B. *Backend* . . . 6
C. API REST . . . 6
D. Python . . . 7
E. Django . . . 7
F. Swagger . . . 8
G. Bases de datos . . . 9
H. PostgreSQL . . . 9
I. Azure *Blob Storage* . . . 10
J. Docker . . . 10
K. Metodologías de desarrollo . . . 11
L. Metodología ágil . . . 11
M. Scrum . . . 12
N. Sistemas de versionamiento de código . . . 12
Ñ. GitHub . . . 13

O. Postman . . . 13
P. Locust . . . 14

6. Metodología **16**
6.1. Levantamiento de requerimientos . . . 16
6.2. *Frameworks*, herramientas y plataformas a utilizar . . . 17
6.3. Diseño de la arquitectura modular . . . 18
6.3.1. Modelo entidad-relación . . . 19
6.3.2. Arquitectura en *Django REST Framework* . . . 22
6.4. Funcionalidades del *backend* . . . 23
6.4.1. Gestión de formularios . . . 24
6.4.2. Gestión de páginas . . . 24
6.4.3. Gestión de campos y clases de campo . . . 24
6.4.4. Gestión de *datasets* externos . . . 24
6.4.5. Gestión de usuarios y autenticación . . . 25
6.4.6. Gestión de asignaciones y permisos . . . 25
6.4.7. Exportación de datos y compatibilidad . . . 25
6.4.8. Automatización mediante señales (*signals*) . . . 25
6.5. Implementación de la infraestructura *backend* . . . 26
6.6. Automatización y despliegue (CI/CD) . . . 27
6.6.1. Objetivo del proceso . . . 27
6.6.2. Arquitectura del flujo . . . 27
6.6.3. Configuración del *workflow* . . . 28
6.6.4. Seguridad y gestión de secretos . . . 29
6.6.5. Estrategia de versionado y trazabilidad . . . 29
6.6.6. Proceso de despliegue . . . 29
6.6.7. Portabilidad del despliegue . . . 29

7. Resultados **30**
7.1. Pruebas funcionales con Postman . . . 30
7.1.1. Objetivo y diseño de las pruebas . . . 30
7.1.2. Resultados obtenidos . . . 31
7.2. Pruebas de seguridad . . . 32
7.2.1. Objetivo y diseño de las pruebas . . . 32
7.2.2. Resultados por escenario . . . 33
7.3. Pruebas de rendimiento y carga con Locust . . . 35
7.3.1. Objetivo y diseño de las pruebas . . . 35
7.3.2. Resultados por escenario . . . 35
7.4. Comparación final . . . 38

8. Conclusiones **40**

9. Recomendaciones **41**

10. Bibliografía **42**

11. Anexos **44**
11.1. Anexo A. Diagramas técnicos . . . 44
11.1.1. Modelo entidad–relación . . . 44
11.1.2. Arquitectura general del sistema . . . 45
11.2. Anexo B. Documentación de la API . . . 45
11.2.1. Autenticación de usuarios . . . 45
11.2.2. Gestión de categorías . . . 45
11.2.3. Gestión de formularios . . . 46
11.2.4. Creación de formularios . . . 46

11.2.5. Gestión de páginas 47
11.2.6. Creación de páginas 47
11.2.7. Gestión de campos 48
11.2.8. Creación de campos 48
11.2.9. Gestión de fuentes de datos 49
11.2.10. Gestión de usuarios 49
11.2.11. Exportación de datos 49

Glosario **50**

Lista de figuras

Figuras **Página**

1. Modelo entidad-relación de la base de datos utilizada para formularios adaptables . . 19
2. Arquitectura lógica del *backend* en *Django REST Framework* 22

3. Ejecución de la colección Postman (CRUD *Flow*) 31
4. Percentiles de tiempo de respuesta (p50 y p95) con 100 usuarios 35
5. *Requests* por segundo (RPS) y fallos por segundo con 100 usuarios 36
6. Percentiles de tiempo de respuesta (p50 y p95) con 80 usuarios 36
7. *Requests* por segundo (RPS) y fallos por segundo con 80 usuarios 37
8. Percentiles de tiempo de respuesta (p50 y p95) con 70 usuarios concurrentes 37
9. *Requests* por segundo (RPS) con 70 usuarios concurrentes 38

10. Modelo entidad–relación del *backend* para formularios adaptables 44
11. Arquitectura modular del sistema *backend* 45
12. *Endpoints* de *login* y *logout* para el uso de rutas por medio de autenticación de usuarios 45
13. *Endpoints* para la gestión de categorías 45
14. *Endpoints* para la gestión de formularios 46
15. *Endpoint* para la creación de formularios 46
16. *Endpoints* para la gestión de páginas de formularios 47
17. *Endpoint* para la creación de páginas dentro de un formulario 47
18. *Endpoints* para la gestión de campos 48
19. *Endpoints* para la gestión de campos 48
20. *Endpoints* para la gestión de fuentes de datos 49
21. *Endpoints* para la gestión de usuarios 49
22. *Endpoints* de exportación de respuestas en formatos Excel, CSV y JSON 49

Lista de cuadros

Cuadros **Página**

1. Flujo CRUD validado con Postman y códigos de estado 32
2. Estadísticas por solicitud en pruebas de seguridad negativa 33
3. Percentiles de tiempo de respuesta en pruebas de seguridad negativa 33
4. Acceso a rutas protegidas sin token 34
5. Acceso a rutas protegidas con token inválido 34
6. Comparación entre Digiforms y la API propia desarrollada 39

Lista de abreviaturas y siglas

CI/CD *Continuous Integration / Continuous Deployment*.

CRUD *Create, Read, Update, Delete*.

HTTP *Status Codes* Códigos de estado HTTP.

JSON JavaScript *Object Notation*.

ORM *Object-Relational Mapping*.

RPS *Requests Per Second*.

SLO *Service Level Objective*.

Resumen

Este trabajo presenta el diseño e implementación de una infraestructura *backend* orientada a la creación, edición y administración dinámica de formularios digitales utilizados en los procesos agrícolas del ingenio. La solución, desarrollada con *Django REST Framework* y PostgreSQL, incorpora módulos que permiten gestionar formularios, páginas y campos de manera flexible, garantizando trazabilidad mediante identificadores únicos y una arquitectura adaptable a cambios futuros.

La base de datos relacional construida para este proyecto almacena de forma íntegra y consistente tanto la estructura de los formularios como la información recolectada en campo, ofreciendo escalabilidad, disponibilidad y control interno sobre los datos. Asimismo, se implementó un sistema de procesamiento y validación capaz de exportar la información en formatos estándar como CSV, Excel y JSON, aplicando reglas de limpieza y normalización para asegurar su calidad.

El *backend* incluye una API REST segura y confiable que sincroniza adecuadamente la aplicación web con la base de datos. Esta API fue evaluada mediante pruebas de autenticación, autorización y carga, demostrando estabilidad incluso bajo niveles de concurrencia superiores a los previstos. En conjunto, la infraestructura desarrollada integra todos los componentes necesarios para garantizar el procesamiento correcto y el almacenamiento confiable de los datos agrícolas capturados en campo, funcionamiento que fue validado mediante pruebas que confirmaron su robustez, estabilidad y desempeño ante formularios adaptables.

Abstract

This work presents the design and implementation of a backend infrastructure for the dynamic creation, editing, and management of digital forms used in the agricultural processes of the sugar mill. The solution, developed with Django REST Framework and PostgreSQL, incorporates modules that enable flexible administration of forms, pages, and fields, ensuring full traceability through unique identifiers and an architecture that can adapt to future changes.

The relational database built for this project stores both the structure of the forms and the field-collected information in an integral and consistent manner, providing scalability, availability, and internal control over the data. Additionally, a data processing and validation system was implemented to export the collected information in standard formats such as CSV, Excel, and JSON, applying cleaning and normalization rules to ensure data quality.

The backend includes a secure and reliable REST API that synchronizes the web application with the database. This API was evaluated through authentication, authorization, and load tests, demonstrating stability even under concurrency levels higher than initially expected. Altogether, the developed infrastructure integrates all components required to guarantee the correct processing and reliable storage of the agricultural data captured in the field functionality that was validated through tests confirming the system's robustness, stability, and performance when handling adaptable forms.

Introducción

La transformación digital en el ámbito agrícola ha impulsado la adopción de soluciones tecnológicas que facilitan la organización, gestión y análisis de la información recolectada en campo. Entre estas soluciones, los formularios digitales con opciones de personalización se han convertido en una herramienta esencial, pues permiten capturar datos de manera flexible y adaptada a las particularidades de cada proceso agrícola. No obstante, esta transición digital no siempre garantiza autonomía ni control total sobre la información recolectada.

En el caso del ingenio, aunque se ha avanzado en la digitalización de los procesos de recolección de datos, persiste una problemática específica: las plataformas disponibles no se ajustan completamente a su necesidad de formularios altamente personalizados. Como solución provisional, actualmente se utiliza Digiforms; sin embargo, esta herramienta almacena los datos en servidores externos y los entrega únicamente después de un periodo determinado. Esta dependencia limita el acceso inmediato a la información, así como la capacidad de administrarla, procesarla o integrarla en tiempo real dentro de los sistemas internos del ingenio.

Esta limitación evidencia la necesidad de desarrollar una plataforma propia que permita diseñar formularios flexibles, gestionar la información sin intermediarios y asegurar un control completo sobre los datos generados en campo.

El presente trabajo se enfoca específicamente en el desarrollo del módulo de infraestructura *backend* de la aplicación web, el cual constituye la base operativa del sistema. Este módulo se encarga de la lógica necesaria para crear, editar y gestionar formularios dinámicos, así como de procesar y almacenar las respuestas enviadas desde la aplicación móvil en una base de datos centralizada. Para ello, se implementa una API REST robusta basada en *Django REST Framework*, diseñada para interactuar con el *frontend* web y con la base de datos encargada de registrar tanto la estructura de los formularios como la información recopilada en campo.

De manera complementaria, se diseña un modelo de base de datos relacional normalizada, orientado a garantizar la integridad, consistencia y escalabilidad de los datos. Esta estructura facilita la trazabilidad de la información, optimiza su disponibilidad y permite exportarla para análisis posteriores en formatos estándar como Excel, CSV o JSON. Gracias a ello, el ingenio obtiene mayor autonomía en el tratamiento de su información y la capacidad de integrarla con herramientas analíticas o procesos internos de toma de decisiones.

Finalmente, se llevaron a cabo pruebas de autenticación, autorización y rendimiento, asegurando que el sistema se mantuviera estable, seguro y funcional incluso bajo cargas concurrentes superiores a las previstas. Esto permitió validar la fiabilidad del *backend* desarrollado y su capacidad para soportar los formularios adaptables utilizados en diversas actividades agrícolas.

Antecedentes

En los últimos años, el ingenio ha dado pasos importantes hacia la modernización de sus procesos operativos, especialmente en lo relacionado con la gestión y recolección de información generada en campo. Las áreas técnicas del ingenio han identificado la necesidad de contar con datos más precisos, oportunos y estructurados, para mejorar actividades como la planificación agrícola, el control de insumos, la ejecución de labores, la trazabilidad de operaciones y la evaluación de resultados productivos. En este contexto, la digitalización de formularios se volvió un componente clave para capturar información de manera más ágil y confiable.

Antes de adoptar soluciones digitales, el ingenio dependía principalmente de registros manuales o herramientas genéricas para documentar las actividades de campo. Este tipo de prácticas dificultaban la estandarización de la información y generaban problemas frecuentes, como errores en la captura, variaciones entre equipos, pérdida de documentos físicos y retrasos en la disponibilidad de los datos para su análisis. Para resolver estas limitaciones, el ingenio decidió implementar de forma temporal una plataforma externa de formularios digitales que permitiera crear formatos, distribuirlos a dispositivos móviles y facilitar su llenado por parte del personal técnico.

Si bien esta plataforma permitió agilizar inicialmente el proceso de captura de datos, también evidenció limitaciones importantes para las necesidades internas del ingenio. Entre ellas destacan: la falta de control directo sobre el almacenamiento de la información, restricciones para adaptar completamente los formularios a los procesos agrícolas propios del ingenio, y tiempos de exportación que no siempre se ajustaban a los requerimientos operativos. Estas condiciones hicieron evidente la importancia de contar con una infraestructura tecnológica propia, diseñada específicamente para los flujos internos de información.

A partir de este diagnóstico, surgió la necesidad de desarrollar un *backend* que permitiera al ingenio gestionar internamente todo el ciclo de vida de sus formularios: desde la creación y edición, hasta la validación, almacenamiento y posterior exportación de los datos. Este enfoque no solo garantiza mayor independencia tecnológica, sino que también ofrece la flexibilidad necesaria para reflejar la estructura real de los procesos agrícolas, asegurar la integridad de la información capturada y disponer de los datos bajo demanda para su análisis o integración con otros sistemas internos.

La implementación de un *backend* propio aporta beneficios directos al ingenio: centraliza toda la información en una plataforma interna, elimina dependencias externas, reduce errores derivados de la digitación manual y mejora la consistencia de los datos. Además, la automatización del flujo de captura y validación permite una operación más ágil y ordenada, aumentando la trazabilidad y la capacidad de respuesta ante necesidades operativas.

En conjunto, estos elementos fundamentan el desarrollo de esta solución tecnológica interna, cuyo objetivo es fortalecer la capacidad del ingenio para recopilar, procesar y analizar información agrícola de manera eficiente, segura y adaptada a sus procesos específicos.

Justificación

En el sector agrícola, la recopilación y gestión de datos juegan un papel fundamental en la toma de decisiones y en la mejora de la productividad. La eficiencia en la recolección de información en campo es crucial para mejorar la planificación, el monitoreo de cultivos y la gestión de recursos. Sin embargo, las soluciones tradicionales suelen presentar limitaciones en cuanto a su flexibilidad y capacidad de adaptación a distintos escenarios, lo que dificulta su implementación en un entorno dinámico y variable como el agrícola.

La digitalización de los procesos de recopilación y gestión de datos permite mejorar la precisión y confiabilidad de la información obtenida en campo, optimizando el uso de recursos y facilitando el análisis posterior. En este sentido, el desarrollo de herramientas digitales adaptables es clave para asegurar que la información recolectada se estructure de manera eficiente y se almacene correctamente en bases de datos relacionales, asegurando su integridad y disponibilidad para análisis posteriores (Gostev et al., 2019).

Ante esta problemática, surge la necesidad de desarrollar una infraestructura *backend* flexible y escalable que permita la creación y gestión de formularios digitales adaptables. Esta solución garantizará la estructuración eficiente de la información recolectada y su correcta integración en una base de datos relacional, permitiendo mantener la integridad y organización de los datos. La implementación de tecnologías digitales en la recopilación de información agrícola no solo facilita la automatización de procesos, sino que también permite la validación de datos antes de su almacenamiento definitivo, reduciendo errores y mejorando la calidad de la información registrada (Gostev et al., 2019).

En base a lo expuesto, en el caso puntual del ingenio, el retraso en el acceso a los datos recolectados mediante plataformas externas como Digiforms representa un obstáculo importante para la toma de decisiones oportuna. Además, la incapacidad de dichas plataformas para ajustarse a los requerimientos particulares de sus operaciones limita su eficacia en un entorno donde cada formulario debe adaptarse a procesos técnicos y agrícolas específicos. Por lo tanto, la implementación de una infraestructura *backend* propia no solo responde a una necesidad tecnológica, sino también estratégica: le permitirá al ingenio tener control completo sobre sus datos, asegurar la disponibilidad inmediata de la información y diseñar formularios personalizados según las necesidades reales del campo. Esta solución fortalecerá sus capacidades internas para la gestión de información, permitirá una integración fluida con sus sistemas y procesos existentes, y sentará las bases para una modernización sostenible y escalable de sus operaciones agrícolas.

4.1. Objetivo general

Desarrollar una infraestructura *backend* para un sistema de gestión de formularios digitales adaptables en el sector agrícola, que garantice el correcto procesamiento y almacenamiento de datos recolectados en campo.

4.2. Objetivos específicos

1. Desarrollar un *backend* flexible para la creación y gestión de formularios dinámicos conforme a las necesidades específicas del usuario administrador.
2. Construir una base de datos relacional normalizada que almacene la estructura de los formularios e información recopilada para garantizar la integridad referencial y la disponibilidad de los datos.
3. Construir un sistema de procesamiento y validación que permita exportar respuestas en distintos formatos como CSV, Excel y JSON, incorporando reglas de limpieza y normalización.
4. Programar una API REST, que sincronice de forma confiable los datos entre la aplicación web y la base de datos empleando controles de autenticación y autorización para permitir la consulta bajo demanda de las respuestas almacenadas.

Marco teórico

Actualmente el ingenio realiza su recopilación de datos en campo con la plataforma web y móvil Digiforms. Esta es una plataforma digital diseñada para facilitar la creación, distribución y recopilación de formularios electrónicos, especialmente útil en contextos donde se requiere recolectar información estructurada en campo o de forma remota. Esta herramienta permite reemplazar los formularios físicos por versiones digitales personalizables, que pueden ser llenadas desde dispositivos móviles o navegadores web, con o sin conexión a internet.

Esta plataforma opera a través de una arquitectura cliente-servidor, combinando una aplicación web para la gestión y diseño de formularios, y una aplicación móvil para su llenado en campo. Este enfoque permite digitalizar procesos de recolección de datos de manera eficiente, incluso en entornos con conectividad limitada. La plataforma contiene:

1. Aplicación web
    - Crear formularios personalizados (con campos de texto, numéricos, *datasets*, etc.).
    - Gestionar usuarios y permisos.
    - Visualizar o exportar datos recolectados.
2. Aplicación móvil
    - Está diseñada para ser utilizada por técnicos o personal de campo.
    - Funciona tanto *online* como *offline*, permitiendo descargar formularios asignados desde el servidor, llenar formularios sin necesidad de conexión a internet y sincronizar los datos recolectados cuando haya acceso a la red.

A. *Framework*

En el ámbito de la ingeniería de software, un *framework* se define como un conjunto de componentes reutilizables que proporciona una base estructurada para el desarrollo de aplicaciones. Estos componentes incluyen bibliotecas de código, herramientas y protocolos que permiten a los desarrolladores construir nuevos sistemas de manera más eficiente al aprovechar módulos ya probados y estandarizados (UNIR, 2022).

El uso de *frameworks* ofrece múltiples ventajas. Como primer punto, mejora la calidad del código, ya que los módulos que lo conforman están desarrollados bajo altos estándares de programación, lo que reduce la probabilidad de errores y facilita la legibilidad. Como segundo punto, contribuye a disminuir los tiempos de desarrollo, pues evita que los equipos deban programar desde cero funciones repetitivas, permitiéndoles concentrarse en la lógica de negocio de sus proyectos. Otra ventaja significativa es la seguridad, ya que muchos *frameworks* incorporan mecanismos preconfigurados para

proteger aplicaciones frente a vulnerabilidades comunes. Asimismo, favorecen la colaboración y mantenibilidad, ya que al imponer un flujo de trabajo estandarizado, distintos desarrolladores pueden comprender e integrar sus aportes más fácilmente.

Un *framework* otorga flexibilidad para introducir cambios o nuevas funcionalidades sin necesidad de reescribir grandes partes del código, lo que incrementa la escalabilidad y sostenibilidad de los proyectos en el tiempo (UNIR, 2022).

B. *Backend*

El *backend* es la parte del desarrollo de software que se encarga de la lógica interna, el procesamiento de datos y la gestión de los recursos que no son visibles para el usuario final. Es responsable de manejar la comunicación entre la interfaz de usuario (*frontend*), la base de datos y otros servicios del sistema (Amazon Web Services, s. f.-b).

Este se implementa generalmente mediante servidores, APIs y sistemas de gestión de bases de datos. Su función principal es recibir, procesar y responder a las solicitudes provenientes del *frontend*, garantizando que la información fluya correctamente, de forma segura y eficiente (Amazon Web Services, s. f.-b).

Actualmente, según Renish (2024), los lenguajes más utilizados para ello son:

- Node.js con JavaSript
- Python

Ambos se han vuelto de los más utilizados, por ejemplo, Node.js por su alto rendimiento y capacidad de manejar muchas conexiones simultáneas, lo que lo hace ideal para aplicaciones en tiempo real, y Python es un lenguaje muy legible y fácil de aprender, ampliamente utilizado en desarrollo web, ya que su sintaxis lo hace ideal para desarrollos rápidos y mantenibles (Renish, 2024).

Algunos de los *frameworks* más comunes según Prytulenets (2025) para estos lenguajes son:

- Django
- Express.js

Express.js es minimalista y flexible, facilita la creación de APIs REST y aplicaciones web. Ahora Django está basado en el patrón Modelo-Vista Controlador (MVC), incluye muchas funcionalidades listas para usar como la autenticación de usuarios, su ORM, etc. A parte fomenta el desarrollo rápido y seguro de aplicaciones web (Prytulenets, 2025).

C. API REST

Una API REST es una interfaz de programación de aplicaciones que sigue los principios del estilo arquitectónico *Representational State Transfer* (REST). Propuesto por Roy Fielding, REST proporciona un marco conceptual para conectar sistemas distribuidos con alto grado de flexibilidad, coherencia, escalabilidad y eficiencia. En la práctica, una API REST expone recursos accesibles mediante HTTP y ofrece una vía ligera para intercambiar datos entre aplicaciones, servicios web y bases de datos, incluidas arquitecturas de microservicios (IBM, 2025b).

Las solicitudes más comunes utilizadas en APIs REST son:

- GET: Solicitud para recuperar información de un recurso sin modificarlo
- POST: Solicitud para enviar datos al servidor y crear un nuevo recurso
- PUT: Solicitud para actualizar completamente un recurso existente o crearlo si no existe.
- DELETE: Solicitud para eliminar un recurso existente

Las representaciones de los recursos pueden serializarse en varios formatos, siendo JSON el más común por su legibilidad humana y compatibilidad entre lenguajes. Una API REST bien diseñada facilita la interoperabilidad entre sistemas heterogéneos, acelera la integración de servicios y soporta escenarios de alta demanda gracias a su carácter stateless, su aprovechamiento de caché y su alineación con estándares abiertos de documentación y seguridad (IBM, 2025b).

D. Python

Python es un lenguaje de programación de alto nivel, interpretado y de propósito general, diseñado para facilitar la lectura y escritura de código mediante una sintaxis clara y estructurada. Más que una herramienta, representa un enfoque de desarrollo orientado a la simplicidad, la expresividad y la productividad, lo que ha impulsado su adopción masiva en múltiples áreas tecnológicas (Worsley, 2024).

Su creación se remonta a finales de los años 80, cuando Guido van Rossum buscó superar las limitaciones del lenguaje ABC, especialmente su falta de extensibilidad. El resultado fue la primera versión de Python en 1991, que ya incluía conceptos clave como orientación a objetos, módulos, manejo de excepciones y estructuras de datos flexibles. Desde entonces, el lenguaje ha evolucionado continuamente, incorporando mejoras sintácticas, soporte Unicode y características avanzadas como *async/await* y tipado más robusto (Worsley, 2024).

Python destaca por su ecosistema amplio de bibliotecas, disponible a través de PyPI, que le permite adaptarse a áreas como desarrollo web, análisis de datos, inteligencia artificial, automatización, finanzas o visualización. *Frameworks* como Django o Flask y paquetes como NumPy, Pandas, TensorFlow o Matplotlib han convertido a Python en una herramienta versátil y esencial para proyectos de distinta naturaleza (Worsley, 2024).

Su impacto también se refleja en la industria: organizaciones como Google, NASA, Spotify, Meta o JP Morgan lo utilizan para servicios backend, análisis masivo de datos, simulaciones, automatización y *machine learning*. Esta adopción generalizada confirma el papel de Python como uno de los lenguajes más influyentes y demandados de la actualidad, combinando accesibilidad con una robusta capacidad técnica (Worsley, 2024).

E. Django

Django es un *framework* web de alto nivel escrito en Python que promueve el desarrollo rápido y el diseño limpio. Ofrece módulos reutilizables para funciones comunes como autenticación, administración de contenidos, acceso a base de datos, gestión de sesiones, etc, de modo que los equipos pueden concentrarse en la lógica de negocio y reducir de forma significativa el tiempo de construcción de aplicaciones web. Además de ser gratuito y de código abierto, cuenta con una comunidad activa y una infraestructura de soporte consolidada, lo que favorece su evolución continua y su adopción en proyectos de distinta escala.

Arquitectónicamente, Django organiza las aplicaciones en el patrón Modelo–Vista–Template (MVT): los modelos encapsulan el acceso a datos y el mapeo objeto–relacional; las vistas orquestan

la lógica de la solicitud y construyen la respuesta; y los *templates* se encargan de la presentación. Este desacoplamiento facilita la mantenibilidad y las pruebas, al tiempo que permite escalar horizontalmente capas específicas como la aplicación o base de datos (Amazon Web Services, s. f.-a).

Django REST Framework (DRF) es un toolkit para construir APIs web sobre Django. Proporciona componentes listos para usar como serializadores y *ModelSerializers* para transformar datos entre modelos y formatos como JSON; clases genéricas de vista y *viewsets* para CRUD estándar; y políticas de autenticación y permisos que aceleran la exposición segura y consistente de servicios REST. En la práctica, DRF se ha convertido en la opción de facto para implementar APIs REST con Django (Django REST Framework, s. f.).

F. Swagger

Swagger es un conjunto de herramientas y especificaciones diseñadas para describir, documentar y consumir API REST de manera estandarizada. Su objetivo principal es permitir que las APIs sean legibles tanto por humanos como por máquinas, mediante un documento estructurado, en formato JSON o YAML, que define los recursos, operaciones, parámetros y respuestas de una API, conforme al estándar OpenAPI Specification (OAS), anteriormente conocido como Swagger Specification (Swagger, s. f.).

El corazón de Swagger radica en su capacidad de autodescripción. Una API que expone un archivo de especificación Swagger permite que otras herramientas comprendan automáticamente su estructura, lo que habilita una amplia gama de posibilidades: generación automática de documentación interactiva, creación de clientes en distintos lenguajes de programación, y ejecución de pruebas automatizadas sin requerir configuración manual (Swagger, s. f.).

La especificación Swagger describe aspectos clave de una API, como:

- Las operaciones y métodos HTTP disponibles (GET, POST, PUT, DELETE, etc.).
- Los parámetros de entrada, tipos de datos y formatos de respuesta.
- Los mecanismos de autorización requeridos.
- Información complementaria como licencias, términos de uso o contactos del desarrollador.

Una de sus principales ventajas es la posibilidad de integrarse directamente con el código fuente. Los desarrolladores pueden generar el archivo de especificación manualmente o mediante herramientas automáticas que extraen información desde las anotaciones del código. Posteriormente, esta especificación puede aprovecharse de diversas formas:

- Swagger UI: genera documentación interactiva en la que los usuarios pueden explorar y probar las operaciones de la API directamente desde un navegador web.
- Swagger Codegen: permite generar automáticamente el stub del servidor o bibliotecas cliente en más de 40 lenguajes, acelerando el desarrollo y la integración.
- SoapUI y otras herramientas: usan la especificación para realizar pruebas automatizadas y validar el cumplimiento del comportamiento esperado.

Gracias a su amplia adopción y compatibilidad con herramientas de código abierto, se ha convertido en un componente esencial en los flujos de desarrollo moderno orientados a servicios (Swagger, s. f.).

G. Bases de datos

En informática, una base de datos es una colección organizada de datos que se almacena y gestiona en un sistema informático para facilitar su acceso, administración y seguridad. Para ello se emplean sistemas de gestión de bases de datos (DBMS), que añaden herramientas de control y gobierno de la información a medida que el volumen y la complejidad de los datos aumentan (Microsoft Azure, s. f.).

Para administrar estos conjuntos de datos, se utilizan los sistemas de gestión de bases de datos, que ofrecen un conjunto de herramientas y funciones para crear, mantener y consultar las bases de datos, asegurando la organización, seguridad y escalabilidad de la información almacenada.

Las bases de datos mayormente utilizadas son las bases de datos relacionales y no relacionales, cada una diseñada para distintos tipos de estructura y requerimientos de acceso a los datos (Microsoft Azure, s. f.).

- Bases de datos relacionales (SQL): Organizan los datos en tablas con filas y columnas, donde cada fila representa un registro único y cada columna un atributo de los datos.
- Bases de datos no relacionales (NoSQL): Diseñadas para manejar grandes volúmenes de datos no estructurados o semiestructurados. Ofrecen esquemas flexibles y son ideales para aplicaciones que requieren escalabilidad horizontal y alta disponibilidad

Los sistemas de gestión de bases de datos relacionales son herramientas que permiten crear, administrar y manipular bases de datos relacionales. Entre los más utilizados según Stackscale (2023) se encuentran:

- MySQL: Sistema de código abierto ampliamente utilizado en aplicaciones web. Destaca por su rendimiento y facilidad de uso.
- PostgreSQL: Sistema avanzado de código abierto que ofrece extensibilidad y cumplimiento de estándares SQL. Es conocido por su robustez y soporte para operaciones complejas.
- Oracle Database: Sistema comercial que ofrece soluciones empresariales con características avanzadas de seguridad y rendimiento.
- Microsoft SQL Server: Sistema desarrollado por Microsoft, integrado con otras herramientas de la empresa, y utilizado en entornos empresariales.

H. PostgreSQL

PostgreSQL es un sistema de gestión de bases de datos relacional (RDBMS) de código abierto, reconocido por su fiabilidad, flexibilidad y cumplimiento con los estándares técnicos abiertos.

A diferencia de otros sistemas de bases de datos relacionales, PostgreSQL soporta tanto tipos de datos relacionales como no relacionales, lo que lo convierte en una herramienta versátil capaz de adaptarse a una amplia variedad de aplicaciones modernas. Entre sus principales ventajas destacan su rendimiento y escalabilidad, optimizados mediante técnicas como la concurrencia sin bloqueos a través del modelo MVCC (*Multiversion Concurrency Control*). Este mecanismo permite que múltiples usuarios accedan simultáneamente a los mismos datos sin interferencias entre lecturas y escrituras, asegurando un desempeño eficiente incluso en sistemas de gran tamaño. Asimismo, PostgreSQL ofrece replicación síncrona y asíncrona, lo que permite garantizar la continuidad del negocio y la alta disponibilidad en entornos críticos (IBM, 2025a).

PostgreSQL también se distingue por su soporte multilenguaje, compatible con entornos de desarrollo en Python, Java, C/C++, JavaScript y Ruby, lo que amplía las posibilidades de integración con distintas plataformas (IBM, 2025a).

I. Azure *Blob Storage*

Azure *Blob Storage* es un servicio de almacenamiento en la nube proporcionado por Microsoft Azure, diseñado para gestionar grandes volúmenes de datos no estructurados, como texto, imágenes, audio, video o archivos binarios. El término blob proviene de *Binary Large Object*, lo que refleja su capacidad de almacenar objetos binarios de tamaño considerable. A diferencia de los sistemas de archivos jerárquicos tradicionales, *Blob Storage* utiliza un modelo de almacenamiento basado en objetos, en el cual los datos se guardan como blobs dentro de contenedores, sin necesidad de una estructura rígida o predeterminada (LogicMonitor, s. f.).

Entre sus principales ventajas se destacan la escalabilidad, la rentabilidad y la accesibilidad global. Al estar integrado en la infraestructura de Azure, el servicio permite manejar cantidades prácticamente ilimitadas de información con una alta disponibilidad y redundancia geográfica. Además, su modelo de precios por niveles (*tiers*) se adapta a las necesidades de uso y frecuencia de acceso (LogicMonitor, s. f.).

Los blobs se organizan dentro de contenedores, los cuales funcionan como unidades lógicas de almacenamiento que permiten una mejor gestión y control de acceso. Un contenedor puede almacenar una cantidad ilimitada de blobs, y las cuentas de almacenamiento pueden incluir múltiples contenedores (LogicMonitor, s. f.).

El servicio es accesible desde cualquier parte del mundo mediante HTTP/HTTPS, a través de la API REST de Azure Storage o mediante bibliotecas cliente disponibles para varios lenguajes de programación, como Python, Java, C#, Node.js o PHP. Además, Azure *Blob Storage* se integra con otras herramientas del ecosistema Microsoft, como Azure *Active Directory* (AAD) para la autenticación y el control de acceso basado en roles (RBAC) o las *Shared Access Signatures* (SAS), que permiten conceder permisos temporales y específicos sobre los datos almacenados (LogicMonitor, s. f.).

En materia de seguridad, todos los blobs se cifran automáticamente antes de ser almacenados y durante su transmisión. Asimismo, se pueden aplicar políticas de acceso mediante el uso de identidades administradas, claves de acceso o tokens de seguridad (LogicMonitor, s. f.).

Su flexibilidad y escalabilidad hacen que Azure *Blob Storage* sea ampliamente utilizado en casos de respaldo y recuperación de datos, análisis de grandes volúmenes de información (*big data*), almacenamiento de archivos multimedia, gestión de registros IoT, y archivado de información corporativa. Gracias a su naturaleza *cloud-native*, los usuarios pueden acceder a sus datos desde cualquier ubicación con conexión a Internet, lo que convierte a Azure *Blob Storage* en una de las soluciones más robustas y versátiles para el almacenamiento de datos empresariales (LogicMonitor, s. f.).

J. Docker

Docker es una tecnología de software que permite la creación y uso de contenedores. Estos contenedores funcionan como unidades ligeras y modulares que aíslan procesos y aplicaciones, ofreciendo un entorno de ejecución consistente y portable. Docker aprovecha funcionalidades del kernel de Linux, como los espacios de nombres y los grupos de control, para dividir procesos y ejecutarlos de manera independiente, lo que permite aprovechar mejor la infraestructura disponible manteniendo altos niveles de seguridad (Red Hat, 2023).

El uso de Docker presenta múltiples ventajas en el desarrollo de software. Una de las más destacadas es la modularidad, ya que facilita dividir aplicaciones en procesos individuales que pueden actualizarse o sustituirse sin afectar al resto del sistema. Además, el modelo de imágenes de Docker, basado en capas y control de versiones, permite agilidad en el desarrollo y la posibilidad de restaurar versiones anteriores con facilidad, lo cual es fundamental en entornos de integración y entrega continua (CI/CD). Otra ventaja clave es la rapidez de implementación: mientras que antes el despliegue de nuevos sistemas podía tardar días, con Docker es posible levantar contenedores en segundos, reduciendo costos y aumentando la eficiencia operativa. Asimismo, al empaquetar una aplicación junto con sus dependencias, Docker asegura que se ejecute de la misma manera en cualquier entorno, evitando problemas comunes de compatibilidad entre fases de desarrollo, pruebas y producción (Red Hat, 2023).

K. Metodologías de desarrollo

Las metodologías de desarrollo de software constituyen marcos de trabajo que guían la planificación, organización y ejecución de proyectos tecnológicos. Su propósito es proporcionar orden, estructura y buenas prácticas en el ciclo de vida del software, desde la definición de requisitos hasta la entrega final del producto. Estas metodologías han evolucionado principalmente en dos enfoques: las metodologías tradicionales y las metodologías ágiles.

Las metodologías tradicionales, como el modelo en cascada (*waterfall*), se caracterizan por un desarrollo lineal y secuencial en el cual cada fase depende de la finalización de la anterior. Bajo este enfoque, los requisitos, el tiempo y el presupuesto se definen desde el inicio y rara vez cambian durante el proceso. Aunque ofrecen un marco probado y predecible, presentan limitaciones frente a proyectos complejos o cambiantes, ya que dificultan la adaptación a nuevas necesidades surgidas durante el desarrollo (Gabaldón, 2021).

En contraste, las metodologías ágiles se basan en ciclos iterativos e incrementales de corta duración. Tras cada iteración, el equipo entrega un incremento funcional del producto, revisado y retroalimentado por el cliente. Esta filosofía enfatiza la flexibilidad, la colaboración estrecha con el usuario y la capacidad de adaptación a cambios en cualquier etapa del proyecto. Entre sus beneficios destacan la reducción de burocracia, mayor eficiencia del equipo, disminución de riesgos y entrega más rápida de valor. Además, han dado lugar a marcos de trabajo ampliamente utilizados como Scrum, Kanban o SAFe, que se aplican no solo en software, sino también en sectores como finanzas, marketing y arquitectura (Gabaldón, 2021).

Tanto las metodologías tradicionales como las ágiles aportan ventajas y desventajas, y la elección de una u otra depende de factores como el tamaño del proyecto, la complejidad de los requisitos y el grado de incertidumbre del entorno. Mientras las tradicionales ofrecen estabilidad y previsibilidad, las ágiles permiten responder con rapidez a las necesidades cambiantes de los usuarios, lo que las convierte en una opción preferida en la mayoría de proyectos de software actuales.

L. Metodología ágil

La metodología ágil es un enfoque de gestión y desarrollo de proyectos que divide el trabajo en fases iterativas, priorizando la entrega continua de valor y la mejora constante. A diferencia de los modelos tradicionales, que avanzan de manera secuencial y rígida, el enfoque ágil promueve la adaptabilidad, la retroalimentación frecuente y la colaboración activa entre los miembros del equipo y con los clientes (Atlassian, s. f.).

Uno de sus pilares fundamentales es la planificación adaptativa, que permite responder con rapidez a cambios en el mercado o a comentarios de los usuarios sin comprometer el proyecto en su totalidad. Esto se logra mediante la entrega de incrementos pequeños y funcionales en plazos cortos,

lo que facilita evaluar avances, reducir riesgos y ajustar las prioridades de manera dinámica. En este contexto, las iteraciones funcionan no solo como ciclos de construcción técnica, sino también como oportunidades de aprendizaje y ajuste continuo (Atlassian, s. f.).

La metodología ágil se centra en las personas y en las interacciones humanas, priorizando la comunicación abierta, la confianza y el trabajo colaborativo por encima de procesos rígidos o de documentación extensa. Esto fomenta la autonomía de los equipos, quienes definen sus propios estándares de calidad y organizan sus tareas de forma auto-gestionada. Como resultado, se incrementa el sentido de compromiso, motivación y responsabilidad compartida, lo que favorece la eficiencia y la calidad de los entregables (Atlassian, s. f.).

M. Scrum

Scrum es un marco de trabajo ágil diseñado para ayudar a los equipos a gestionar proyectos de manera colaborativa mediante ciclos cortos llamados *sprints*. Se centra en aplicar los principios de la metodología ágil a través de un conjunto de prácticas, roles y artefactos específicos que permiten organizar el trabajo, revisar los avances y adaptarse continuamente a los cambios (Microsoft Learn, 2023).

El ciclo de vida de Scrum se organiza en *sprints* con una duración de entre una y cuatro semanas. Cada *sprint* incluye planificación, ejecución, revisión y retrospectiva, con el objetivo de entregar un incremento del producto que cumpla con los criterios de calidad establecidos. Esta iteración constante facilita la retroalimentación temprana y la mejora continua, reduciendo riesgos y permitiendo al equipo aprender y adaptarse rápidamente (Microsoft Learn, 2023).

Dentro de Scrum se definen tres roles principales: el *Product Owner*, responsable de priorizar y gestionar el trabajo pendiente, el *Scrum Master*, encargado de asegurar que se cumpla el proceso y de eliminar impedimentos y el equipo de desarrollo, que construye y garantiza la calidad del producto. Además, Scrum introduce prácticas como el Scrum diario, una reunión breve que promueve la transparencia y coordinación entre los miembros del equipo, y artefactos como el panel de tareas y el gráfico de evolución, que permiten visualizar el progreso y planificar mejor la carga de trabajo (Microsoft Learn, 2023).

Otra característica clave es la revisión y la retrospectiva al final de cada *sprint*. De esta manera, Scrum fomenta un entorno de mejora continua, en el que la transparencia y la comunicación abierta son elementos centrales (Microsoft Learn, 2023).

N. Sistemas de versionamiento de código

Un sistema de control de versiones (VCS) es una herramienta fundamental en el desarrollo de software que permite registrar, gestionar y coordinar todos los cambios realizados en el código fuente y otros archivos relacionados con un proyecto. Su principal función es llevar un historial detallado de modificaciones, indicando quién realizó cada cambio, en qué momento y con qué propósito. Esto ofrece la posibilidad de regresar a versiones anteriores en caso de errores, así como de analizar la evolución completa del proyecto (GitLab, s. f.).

El control de versiones protege el código de pérdidas o daños irreversibles y otorga a los equipos la libertad de experimentar sin temor a comprometer la integridad del proyecto. Además, fomenta la colaboración al permitir que múltiples desarrolladores trabajen simultáneamente sobre el mismo código, identificando y resolviendo conflictos de manera eficiente. Gracias a este sistema, los equipos pueden comparar versiones y fusionar cambios (GitLab, s. f.).

Entre sus principales beneficios destacan la calidad, al fomentar revisiones de código entre pares que elevan los estándares de desarrollo; la aceleración, ya que las ramas y fusiones permiten trabajar en paralelo, reduciendo los tiempos de entrega; y la visibilidad, puesto que el repositorio actúa como un registro centralizado y transparente de la evolución del *software*. Todo ello convierte a los sistemas de versionamiento en un componente indispensable de las prácticas modernas de *DevOps* y en un pilar para la integración y entrega continua (CI/CD) (GitLab, s. f.).

Ñ. GitHub

GitHub es una plataforma de desarrollo colaborativo que se basa en el sistema de control de versiones distribuido Git, creado por Linus Torvalds en 2005 para gestionar el desarrollo del kernel de Linux. Mientras que Git es la herramienta que permite registrar y gestionar los cambios en el código fuente, GitHub actúa como un servicio en la nube que aloja repositorios y añade funcionalidades orientadas a la colaboración, la revisión de código y la gestión de proyectos de *software* (GeeksforGeeks, 2025).

GitHub proporciona un entorno en el que los equipos pueden clonar repositorios, crear ramas para nuevas funcionalidades, realizar confirmaciones de cambios y, posteriormente, integrarlos mediante fusiones. Además, la plataforma ofrece herramientas que facilitan la colaboración, como las *pull requests*, revisiones de código en línea, seguimiento de incidencias y tableros de proyectos (GeeksforGeeks, 2025).

Entre sus principales ventajas destacan:

- Historial detallado de cambios que facilita la identificación de errores y la restauración a versiones estables.
- Gestión de ramas que permite el desarrollo paralelo de nuevas características sin afectar el código principal.
- Integración con flujos de *DevOps*, gracias a servicios como GitHub *Actions*, que permiten la automatización de pruebas e implementaciones.
- Acceso universal, al ser una plataforma en línea que funciona como un repositorio centralizado y confiable del proyecto.

O. Postman

Postman es una herramienta de colaboración y desarrollo utilizada para probar, documentar y analizar servicios web y aplicaciones que exponen interfaces de programación de aplicaciones (APIs). Proporciona una interfaz gráfica de usuario (GUI) intuitiva y sencilla que permite a los desarrolladores enviar solicitudes HTTP o HTTPS a servidores web y recibir las respuestas correspondientes de manera visual y estructurada (Formadores IT, 2023).

Esta plataforma facilita la interacción con APIs durante todo su ciclo de vida, ya que permite crear, organizar y ejecutar pruebas automatizadas para verificar el correcto funcionamiento de los *endpoints*. Además, posibilita gestionar diferentes entornos de desarrollo, por ejemplo, desarrollo, pruebas y producción, adaptando fácilmente los parámetros, encabezados o credenciales según cada contexto (Formadores IT, 2023).

Entre sus principales características destacan:

- **Envío de solicitudes:** permite ejecutar peticiones de distintos métodos HTTP como `GET`, `POST`, `PUT`, `PATCH` o `DELETE`, configurando encabezados, parámetros y cuerpos de solicitud de manera personalizada.
- **Colecciones de solicitudes:** posibilita agrupar conjuntos de pruebas o *endpoints* relacionados dentro de colecciones, facilitando la organización, reutilización y colaboración entre miembros del equipo.
- **Gestión de entornos:** ofrece la opción de definir variables globales o de entorno que permiten cambiar fácilmente entre configuraciones de desarrollo, pruebas o producción sin modificar las solicitudes manualmente.
- **Pruebas automatizadas:** integra un motor de *scripting* basado en JavaScript que permite validar automáticamente las respuestas, detectar errores y garantizar la calidad del servicio.
- **Documentación de APIs:** genera documentación detallada y actualizada de forma automática a partir de las solicitudes y respuestas ejecutadas, lo cual mejora la comprensión y el uso de la API por parte de otros desarrolladores.

Entre las ventajas más relevantes de Postman se encuentran su facilidad de uso, compatibilidad con múltiples tecnologías y protocolos web (HTTP, HTTPS, REST, , GraphQL), y su capacidad para integrarse con herramientas de desarrollo como GitHub, Swagger o Jenkins. Asimismo, fomenta la colaboración entre los miembros del equipo mediante el intercambio de colecciones y entornos, permitiendo mantener una comunicación fluida y unificada durante el desarrollo del sistema (Formadores IT, 2023).

P. Locust

Locust es una herramienta de código abierto utilizada para realizar pruebas de rendimiento y carga sobre aplicaciones web, servicios HTTP u otros protocolos. Su principal característica es que permite definir escenarios de prueba utilizando código Python estándar, lo que la hace flexible, escalable y fácilmente integrable con entornos de desarrollo existentes (Locust Documentation, 2025).

A diferencia de otras herramientas que requieren configuraciones complejas o interfaces gráficas limitadas, Locust adopta un enfoque programático. Las pruebas se describen mediante *scripts* de Python en los que se define el comportamiento de los usuarios simulados (denominados *locusts*), permitiendo estructurar bucles, condiciones, cálculos y secuencias personalizadas. Cada usuario virtual se ejecuta dentro de un *greenlet*, una corrutina ligera que posibilita simular miles de conexiones concurrentes con un consumo mínimo de recursos (Locust Documentation, 2025).

Entre sus características más relevantes se destacan:

- **Escenarios de prueba en Python:** Los usuarios y sus acciones se modelan mediante clases y métodos escritos en Python puro, permitiendo controlar la lógica, el flujo y las condiciones de ejecución sin depender de lenguajes de dominio específicos o archivos XML.
- **Ejecución distribuida y escalable:** Admite la ejecución de pruebas de carga distribuidas en múltiples máquinas, pudiendo simular cientos de miles de usuarios simultáneos gracias a su arquitectura basada en eventos.
- **Interfaz web intuitiva:** Proporciona una interfaz gráfica desde la cual se pueden iniciar, monitorear y ajustar las pruebas en tiempo real, visualizando métricas como el número de usuarios activos, tasa de solicitudes, tiempos de respuesta y errores.

- **Compatibilidad con CI/CD:** puede ejecutarse en modo sin interfaz gráfica, lo que facilita su integración en procesos de integración continua y entrega continua, automatizando las pruebas de carga dentro del ciclo de desarrollo.
- **Extensibilidad y personalización:** gracias a su arquitectura modular, permite la incorporación de bibliotecas de Python, el envío de datos a sistemas externos o la creación de clientes personalizados para probar protocolos no estándar.

Metodología

La metodología empleada en este proyecto se fundamentó en un enfoque ágil, específicamente bajo el marco de trabajo Scrum, que permitió organizar el desarrollo del *backend* en fases iterativas y progresivas. Esta elección respondió a la necesidad del ingenio de contar con una solución flexible, escalable y que pudiera ajustarse a nuevas necesidades en el transcurso del proyecto.

El proceso metodológico se estructuró en siete apartados: el levantamiento de requerimientos, la definición de *frameworks* y herramientas a utilizar, el diseño de la arquitectura modular, la especificación de las funcionalidades del *backend*, la implementación de la infraestructura, la ejecución de pruebas de la API y finalmente, la automatización y despliegue de la misma.

6.1. Levantamiento de requerimientos

El análisis y levantamiento de requerimientos representa la etapa inicial de un proyecto de desarrollo de software y constituye la base sobre la cual se construyen las fases posteriores. Su propósito principal es identificar, recopilar y documentar las necesidades de los usuarios y de las partes interesadas, con el fin de establecer qué funcionalidades y características debe ofrecer el sistema. En esta etapa se buscó comprender de manera clara la situación actual del ingenio y las limitaciones de las plataformas que estaban utilizando para la recopilación de datos.

El ingenio empleaba Digiforms como solución provisional, herramienta que si bien permitía digitalizar formularios, presentaba restricciones significativas cómo:

- Dependencia de servidores externos.
- Retraso en la entrega de información recolectada.
- Complicaciones en la personalización de la estructura de formularios.

Con base en estas limitaciones, se identificaron los siguientes requerimientos clave:

- Un módulo *backend* propio, capaz de generar formularios digitales totalmente personalizables.
- Una base de datos relacional para almacenar formularios, páginas, campos y *datasets* que funcione como integración con la aplicación móvil, de manera que esta pudiera consumir la estructura de los formularios creados y pueda almacenar sus respuestas.

- Un módulo de exportación de datos, que facilitara la obtención de las respuestas almacenadas en formatos como Excel, CSV y JSON.
- La posibilidad de cargar *datasets* externos en formato Excel para ser utilizados como listas de autocompletado en campos específicos.

Para obtener y validar estos requerimientos se realizaron reuniones periódicas con los representantes del ingenio. Adicionalmente, se analizaron los procesos internos relacionados con la recopilación de datos agrícolas, lo que permitió asegurar que el diseño del sistema respondiera a necesidades reales y no solamente teóricas.

6.2. *Frameworks*, herramientas y plataformas a utilizar

El desarrollo del módulo *backend* se apoyó en un conjunto de *frameworks*, bibliotecas y servicios que facilitan la construcción de una arquitectura moderna, segura y escalable. La selección de estas herramientas se basó en su madurez, documentación, compatibilidad con Python y capacidad de integrarse de forma eficiente con otros sistemas. A continuación, se detallan las principales tecnologías utilizadas:

- **Lenguaje de programación: Python 3.11**
  Python fue seleccionado como lenguaje principal debido a su sintaxis clara y la gran variedad de librerías disponibles para el desarrollo web, manejo de datos y automatización. Su integración con *frameworks* como Django facilita la implementación de aplicaciones orientadas a servicios y el manejo de estructuras relacionales complejas.
- ***Framework* web: Django 5.x**
  Django proporciona una estructura robusta para el desarrollo rápido de aplicaciones web seguras y escalables. Su ORM (*Object Relational Mapper*) simplifica la interacción con bases de datos relacionales, garantizando integridad referencial y facilitando la manipulación de datos mediante modelos. Además, su sistema de autenticación integrado y manejo de migraciones permiten mantener un control preciso sobre la evolución del esquema de datos.
- ***Framework* de API: *Django REST Framework* (DRF)**
  DRF se utilizó para la construcción de la API RESTful que comunica el *backend* con la base de datos. Este *framework* provee herramientas para la serialización de datos, manejo de permisos, autenticación mediante tokens y generación de *endpoints*, promoviendo un diseño modular y reutilizable. La API implementa operaciones CRUD completas para las entidades del sistema (formularios, páginas, campos, usuarios y *datasets*).
- **Base de datos relacional: PostgreSQL**
  La infraestructura fue diseñada para ser compatible con bases de datos SQL, siendo PostgreSQL el que se va a utilizar por su estabilidad, soporte a datos JSON y extensiones avanzadas. El modelo entidad-relación estructurado permite versionar formularios, almacenar respuestas y mantener relaciones jerárquicas entre entidades como formularios, páginas y campos personalizados definidos por los usuarios.
- **Gestión de almacenamiento de archivos: Azure Blob Storage**
  Se integró la plataforma Microsoft Azure *Blob Storage* para almacenar y administrar los archivos externos utilizados en campos tipo *dataset*. A través del servicio `AzureBlobStorageService`, los archivos Excel o CSV se suben, validan y previsualizan de forma segura. Esta integración permite manejar grandes volúmenes de datos de autocompletado sin sobrecargar el servidor principal.

- **Sistema de autenticación: OAuth2 y Django OAuth Toolkit**
  Para garantizar la seguridad de acceso, se implementó el protocolo OAuth2, mediante el uso de la librería `django-oauth-toolkit`. Este sistema permite la emisión, validación y revocación de tokens de acceso, asegurando que solo usuarios autenticados puedan interactuar con la API. Adicionalmente, se emplean permisos personalizados que limitan las acciones según el tipo de usuario y su nivel de acceso (web o móvil).

- **Control de versiones: Git y GitHub**
  Se utilizó Git como sistema de control de versiones distribuido para gestionar los cambios en el código fuente y coordinar el trabajo colaborativo. El repositorio se alojó en GitHub, lo que permitió la implementación de flujos de trabajo mediante ramas (*branches*) y la integración con GitHub Actions para la automatización de pruebas y despliegues.

- **Metodología ágil: Scrum**
  El desarrollo se organizó mediante la metodología ágil Scrum, estructurada en iteraciones o *sprints* semanales. Este enfoque facilitó la entrega progresiva de funcionalidades, la revisión continua con el ingenio y la incorporación oportuna de mejoras. En cada *sprint* se definieron tareas específicas, asegurando la trazabilidad de los avances.

- **Entorno de desarrollo y pruebas: Visual Studio Code, Postman y Locust**
  Se utilizó Visual Studio Code como entorno principal de desarrollo por su integración con Git, compatibilidad con entornos virtuales de Python y extensiones para depuración.

  Postman se empleó para la documentación y prueba de los *endpoints* REST, permitiendo validar la correcta comunicación entre el *backend* y los clientes externos.

  Finalmente Locust se utilizó para realizar pruebas de carga y seguridad, dándonos un estimado de hasta cuantos usuarios pueden usar la plataforma al mismo tiempo que todos los *endpoints* se encuentren asegurados con el método de autenticación empleado.

- **Librerías complementarias: Argon2 y Django signals**

  - Argon2: Algoritmo implementado para el cifrado seguro de contraseñas de usuarios, garantizando la protección de credenciales mediante un esquema resistente a ataques de fuerza bruta.
  - Django *signals*: Empleado para ejecutar acciones automáticas tras ciertos eventos, como la creación de formularios o la revocación de tokens al desactivar usuarios.

El conjunto de estas herramientas permitió desarrollar un *backend* flexible, escalable y seguro, capaz de integrarse con el *frontend* web y la base de datos para manejar datos estructurados provenientes del sector agrícola. La combinación de *Django REST Framework*, OAuth2 y Azure *Blob Storage* conforma una infraestructura moderna orientada a la interoperabilidad, la trazabilidad y la gestión confiable de información.

6.3. Diseño de la arquitectura modular

Definidos los requerimientos, se procedió al diseño de la arquitectura modular que guiara la implementación del sistema. La premisa principal fue garantizar la escalabilidad y flexibilidad, de manera que el sistema pudiera ampliarse en el futuro sin comprometer su estabilidad.

6.3.1. Modelo entidad-relación

El modelo entidad–relación (MER) se diseñó para soportar formularios dinámicos, versionamiento estructural y trazabilidad de respuestas. La arquitectura propuesta implementa un esquema relacional normalizado que garantiza la integridad referencial y permite la evolución controlada de las estructuras de captura. La Fig. 1 presenta las entidades principales y sus relaciones.

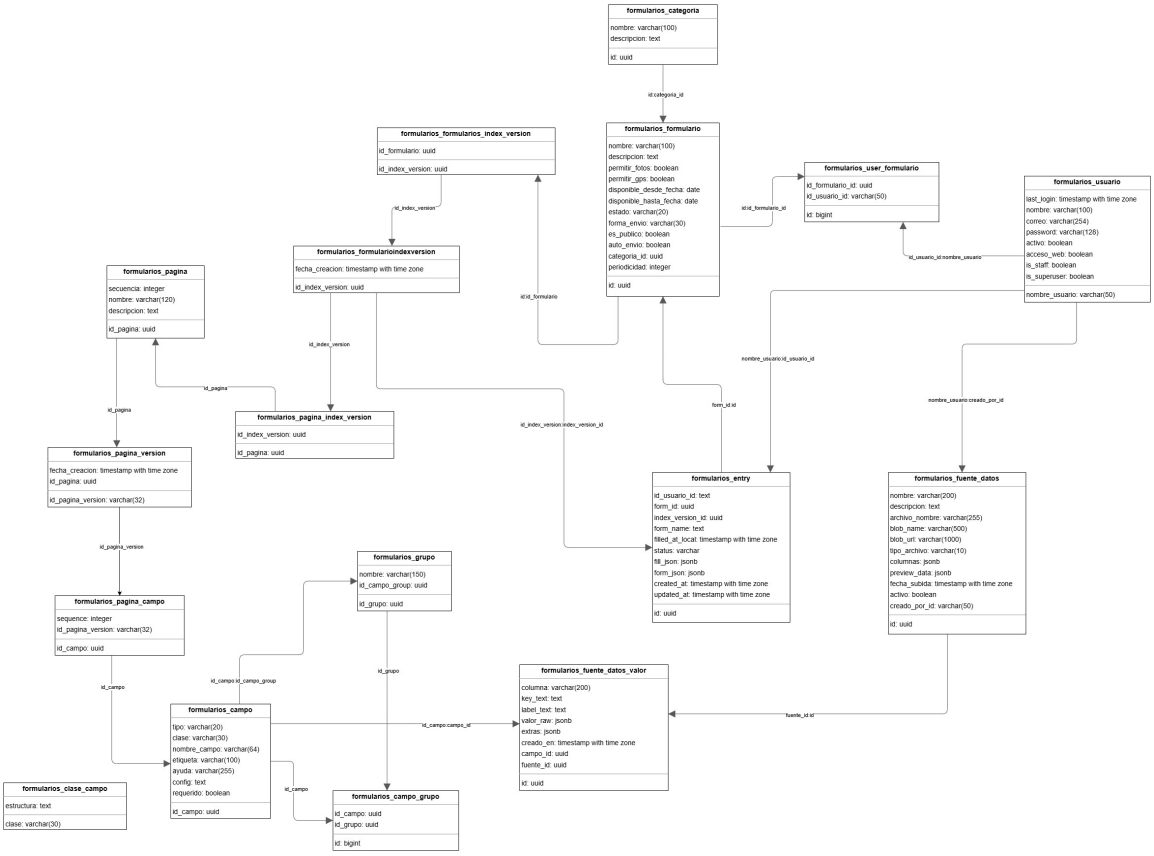


Figura 1: Modelo entidad-relación de la base de datos utilizada para formularios adaptables

Entidades principales y su función en el sistema

Gestión de formularios

- **formularios_categoria**: Taxonomía jerárquica para organizar formularios según su propósito.
- **formularios_formulario**: Entidad central que define la estructura general de cada formulario. Almacena:
  - Metadatos descriptivos (nombre, descripción, categoría)
  - Control de estado (borrador, publicado, archivado)
  - Ventanas temporales de disponibilidad (`fecha_desde`, `fecha_hasta`)
  - Configuración de comportamiento (`auto_envio`, `público`, `periodocidad`)
  - UUID para identificación única entre versiones

- **formularios_pagina**: Agrupaciones lógicas que segmentan el formulario en secciones navegables. Cada página mantiene:
  - Secuencia de presentación (orden numérico)
  - Metadatos de navegación (descripción textual)
  - Relación jerárquica con el formulario padre

Estructura de campos y validaciones

- **formularios_campo**: Define cada elemento de captura con:
  - Tipificación de datos (*dataset*, grupo, numérico, etc.)
  - Configuración de validación, por ejemplo si es requerido.
  - Identificación del campo, como `nombre_campo`, etiqueta, etc.
  - Configuración extendida mediante campo JSON (`config`)
- **formularios_campo_grupo**: Especialización para campos de tipo grupo que permite:
  - Anidación de subcampos (relación con `formularios_grupo`)
- **formularios_grupo**: Contenedor lógico para campos relacionados que:
  - Agrupa campos mediante `id_campo_group`

Gestión de datos externos

- **formularios_fuente_datos**: Repositorio de *datasets* externos con:
  - Identificación descriptiva del documento (nombre, descripción)
  - Link del almacenamiento del archivo fuente en Azure
  - Mapeo de columnas disponibles
  - Datos preprocesados (`preview_data`, `fecha_subida`)
- **formularios_fuente_datos_valor**: Vista normalizada que:
  - Descompone *datasets* en pares columna-valor
  - Facilita vinculación con campos de autocompletado
  - Optimiza consultas de búsqueda mediante índices
  - Mantiene integridad referencial con fuente padre

Control de versiones y respuestas

- **formularios_index_version**: Registro de evolución estructural que:
  - Preserva *snapshots* de configuración (`id_formulario_uuid`)
  - Mantiene linaje de versiones (`id_index_version`)
  - *Timestamp* de creación
- **formularios_entry**: Registro de respuestas a los formularios que contiene:
  - Vínculo usuario-formulario-respuesta
  - Marcas temporales de gestión (`created_at`, `updated_at`)
  - Estado del proceso de captura
  - Registro de respuestas por campo del formuario (`fill_json`)

Gestión de usuarios y permisos

- **formularios_usuario**: Registro de actores del sistema con:
  - Credenciales de acceso (nombre, correo, *password* con hash)
  - Marcas temporales de actividad (`last_login`)
  - Estado de cuenta (active: `boolean`)
  - Roles y permisos (`is_staff`, `is_superuser`)
- **formularios_user_formulario**: Matriz de permisos que:
  - Define acceso granular por formulario
  - Permite asignación dinámica de formularios
  - Soporta múltiples formularios por usuario

Relaciones y cardinalidades

El modelo implementa las siguientes relaciones críticas:

- **1:N entre formulario y páginas**: Un formulario contiene múltiples páginas ordenadas secuencialmente.
- **1:N entre página y campos**: Cada página agrupa varios campos de captura.
- **1:N entre grupo y campo_grupo**: Un campo de clase grupo puede contener múltiples subcampos.
- **M:N entre usuario y formulario**: Implementada mediante la tabla intermedia `user_formulario`.
- **1:N entre formulario y entry**: Un formulario genera múltiples instancias de respuesta.

Consideraciones de diseño

La arquitectura del MER incorpora los siguientes principios de diseño:

- **Normalización**: Tercera forma normal para eliminar redundancias y anomalías de actualización.
- **Flexibilidad**: Campos JSON (`config`, `preview_data`) para extensibilidad sin modificación de esquema.
- **Versionamiento**: Preservación de estados históricos mediante `index_version`.
- **Escalabilidad**: Índices en claves foráneas y campos de búsqueda frecuente.
- **Integridad**: *Constraints* de clave foránea con cascada controlada.
- **Auditoría**: *Timestamps* automáticos en entidades críticas (`created_at`, `updated_at`).

6.3.2. Arquitectura en *Django REST Framework*

La infraestructura *backend* se desarrolló utilizando *Django REST Framework* (DRF), que permitió estructurar el sistema bajo una arquitectura modular basada en servicios web REST. Cada componente del *backend* se implementó mediante vistas, modelos y serializadores independientes, promoviendo la mantenibilidad, la reutilización de código y la escalabilidad de la plataforma.

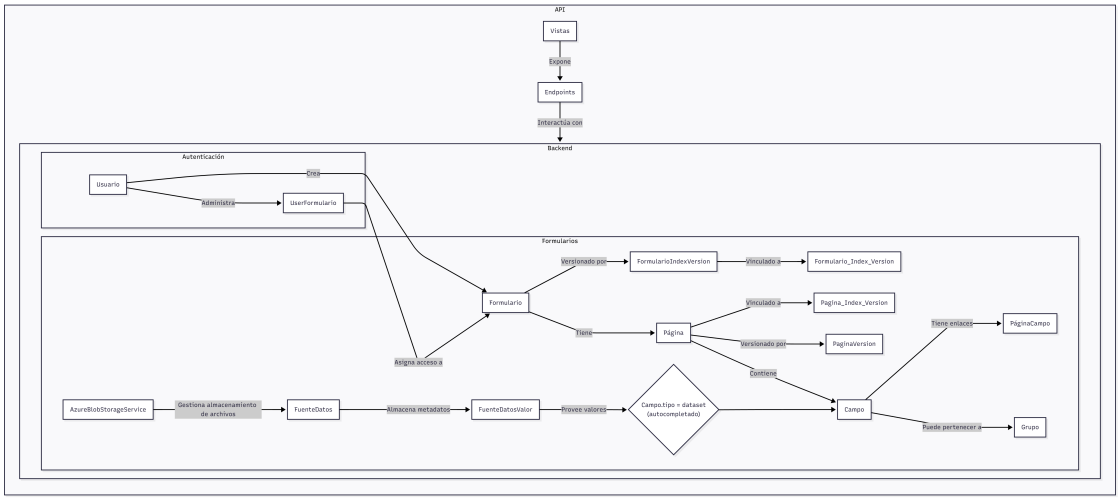


Figura 2: Arquitectura lógica del *backend* en *Django REST Framework*

El diseño sigue un patrón de arquitectura en capas, donde las principales responsabilidades se distribuyen entre:

- **Modelos (*models*):** representan la estructura de la base de datos relacional, permitiendo mapear las entidades principales del sistema, como *Formulario*, *Página*, *Campo*, *Grupo*, *Fuente de Datos* y *Usuario*. Cada modelo está diseñado con claves primarias tipo UUID para asegurar unicidad global y trazabilidad a lo largo de las versiones del sistema.
- **Serializadores (*serializers*):** definen la conversión entre los objetos del modelo y los formatos JSON utilizados por la API, manejando validaciones, relaciones y transformaciones de datos complejas, como la normalización de configuraciones para campos de tipo *dataset*.
- **Vistas (*viewsets*):** exponen los *endpoints* RESTful del sistema, implementando operaciones CRUD (crear, leer, actualizar y eliminar) para las entidades principales. Cada conjunto de vistas se organiza bajo un *viewset*, lo que facilita la integración automática con el enrutador de *Django REST Framework* y la generación de rutas estandarizadas.
- **Servicios (*services*):** encapsulan la lógica de negocio más compleja, como el versionamiento de formularios y páginas, la creación de campos dinámicos, la vinculación de campos a grupos, la gestión de *datasets* externos y el hash seguro de contraseñas.

La plataforma se diseñó bajo un esquema de rutas REST modulares que permiten operar cada componente de forma independiente, garantizando la extensibilidad y separación de responsabilidades. Entre las principales rutas implementadas se encuentran:

- **Rutas de formularios:** Gestionan la creación, edición, duplicación, suspensión y eliminación. Cada formulario puede tener múltiples páginas y versiones, lo que permite mantener un historial estructural sin perder información previa.

- **Rutas de páginas:** Permiten la creación y modificación de secciones dentro de un formulario, así como la adición de campos en cada página mediante *endpoints* especializados. Cada nueva página se asocia automáticamente a la versión activa del formulario.
- **Rutas de campos:** Manejan la definición y administración de los distintos tipos de campos (*text*, *number*, *boolean*, *group*, *dataset*, etc.). La lógica de creación incluye la normalización de configuraciones, el control de secuencias y la vinculación automática con los grupos cuando aplica.
- **Rutas de fuentes de datos:** Permiten la carga, validación y previsualización de archivos Excel o CSV almacenados en Azure *Blob Storage*. Este módulo se integra con el servicio `AzureBlobStorageService`, que maneja la subida, lectura y eliminación de archivos, así como la generación de vistas previas (*previews*) para campos de autocompletado tipo *dataset*.
- **Rutas de usuarios:** Gestionan la creación, actualización y permisos de acceso de los usuarios registrados en el sistema. Se implementó un modelo de autenticación personalizado (`Usuario`) basado en el sistema de usuarios de Django, adaptado para incluir banderas como `activo` y `acceso_web`, controladas mediante permisos personalizados.
- **Rutas de autenticación:** Se implementó un sistema OAuth2 completo para la emisión, validación y revocación de tokens de acceso, garantizando la seguridad y el control de sesiones tanto para usuarios web como para clientes móviles. Los *endpoints* de *login*, *logout* y *user info* permiten la integración segura con aplicaciones externas.
- **Rutas de exportación:** posibilitan la descarga de los datos recolectados en distintos formatos (Excel, CSV, JSON), facilitando el procesamiento posterior y la interoperabilidad con herramientas analíticas.

El proceso de versionamiento de formularios y páginas constituye un eje central de la arquitectura. Cada vez que se crea o modifica un formulario o uno de sus componentes, se genera una nueva versión en las tablas `FormularioIndexVersion` y `PaginaVersion`, las cuales permiten mantener trazabilidad de los cambios sin sobrescribir las versiones anteriores. Esto resulta esencial para garantizar la consistencia entre las versiones de los formularios utilizados en campo y las almacenadas en la base de datos.

Asimismo, el módulo de gestión de *datasets* integra validaciones automáticas y versionamiento interno a través de las tablas `FuenteDatos` y `FuenteDatosValor`, que aseguran la integridad de los valores utilizados en campos de autocompletado. Esta información se almacena y actualiza dinámicamente a partir de los archivos subidos a Azure, permitiendo mantener actualizados los catálogos sin intervención manual.

Finalmente, para mantener la seguridad e integridad de los datos, se implementaron mecanismos automáticos mediante señales (*signals*) que revocan los tokens de autenticación de usuarios cuando se desactivan sus permisos, y que crean versiones iniciales de formularios en el momento de su registro. De esta manera, la arquitectura no solo garantiza modularidad y trazabilidad, sino también robustez y seguridad en la gestión de la información.

6.4. Funcionalidades del *backend*

El *backend* desarrollado constituye el núcleo lógico de la plataforma de formularios adaptables, permitiendo gestionar toda la información estructural, operativa y de seguridad del sistema. Su arquitectura modular en *Django REST Framework* integra diversas funcionalidades que garantizan la creación dinámica de formularios, la administración de versiones, la gestión segura de usuarios, y la interoperabilidad con aplicaciones móviles y fuentes de datos externas. A continuación, se describen las funcionalidades más relevantes implementadas:

6.4.1. Gestión de formularios

El módulo de formularios constituye el componente principal del sistema, encargado de la creación, edición, duplicación, suspensión y eliminación de formularios digitales. Cada formulario está asociado a una categoría y cuenta con propiedades configurables como fechas de disponibilidad, visibilidad pública y modo de envío (en línea o fuera de línea). Además, el sistema incorpora un mecanismo de versionamiento estructural, mediante las tablas `FormularioIndexVersion` y `Formulario_Index_Version`, que permiten conservar el historial completo de modificaciones sin sobrescribir versiones anteriores. Este enfoque asegura trazabilidad y consistencia de los datos que se almacenen recolectados en campo.

6.4.2. Gestión de páginas

Cada formulario se compone de una o varias páginas, las cuales actúan como secciones lógicas que agrupan campos relacionados. El sistema permite crear, modificar y eliminar páginas de forma dinámica, manteniendo un control de orden mediante un atributo de secuencia. Cada nueva página se asocia automáticamente con la versión activa del formulario.

6.4.3. Gestión de campos y clases de campo

El módulo de campos permite definir los elementos que componen cada formulario, ofreciendo soporte para múltiples tipos de datos: texto, numéricos, booleanos, listas, fechas, horas, firmas digitales, cálculos automáticos, grupos repetibles y campos con autocompletado basados en *datasets* externos. Cada campo se registra en la tabla `Campo`, donde se almacena su tipo, clase, nombre, etiqueta, ayuda contextual, configuración y si es requerido o no. Durante su creación, se asigna automáticamente una secuencia dentro de la página y se actualizan los punteros de versión correspondientes. Los campos se gestionan a través de *endpoints* dedicados que permiten:

- Crear campos individuales dentro de una página específica.
- Editar configuraciones de campos existentes, ya sea reemplazando o fusionando configuraciones previas.
- Consultar los campos asociados a una página o a un grupo determinado.
- Eliminar campos obsoletos o sin enlace activo.

El sistema también soporta campos de tipo **grupo**, que permiten crear subconjuntos de campos repetibles, por ejemplo, múltiples registros de muestras dentro de un mismo formulario. Esta funcionalidad se implementa mediante las tablas `Grupo` y `CampoGrupo`, que definen las relaciones jerárquicas entre campos contenedores y campos hijos.

6.4.4. Gestión de *datasets* externos

Otra de las funcionalidades más destacadas del *backend* es el manejo de *datasets*, que permiten integrar listas de valores externos en campos de autocompletado o selección. Los archivos cargados por los usuarios (en formato Excel o CSV) se almacenan en la nube mediante Azure *Blob Storage*, gestionados a través del servicio `AzureBlobStorageService`. Este servicio ofrece operaciones de carga, descarga, eliminación y previsualización de archivos, garantizando un almacenamiento seguro y escalable. Los metadatos de cada *dataset* se gestionan mediante la tabla `FuenteDatos`, lo que

permite conservar el historial de cambios y actualizar los catálogos de valores sin afectar la integridad de los formularios ya desplegados. Cada vez que un campo de tipo *dataset* es creado o actualizado, el sistema:

1. Valida el archivo cargado y genera una vista previa con las primeras filas.
2. Crea o reutiliza una versión en la base de datos y materializa los valores correspondientes.

6.4.5. Gestión de usuarios y autenticación

El sistema de usuarios se basa en un modelo personalizado (`Usuario`) que extiende el sistema de autenticación de Django para incluir atributos adicionales como `activo` y `acceso_web`. Se implementó un control granular de accesos mediante el permiso `IsWebAllowed`, que restringe el uso de la plataforma web únicamente a usuarios habilitados, diferenciando a los usuarios de la aplicación móvil. La autenticación se gestiona a través del protocolo OAuth2, implementado mediante `django-oauth-toolkit`. Los *endpoints* de `login`, `logout` y `user_info` permiten el control seguro de sesiones, emisión y revocación de tokens, y protección de los *endpoints* con autenticación basada en tokens tipo *Bearer*. Adicionalmente, se integraron señales automáticas que revocan los tokens activos cuando un usuario es desactivado o pierde permisos de acceso, reforzando así la seguridad del sistema.

6.4.6. Gestión de asignaciones y permisos

El *backend* incluye un módulo de asignaciones que permite vincular usuarios con formularios específicos, controlando qué formularios pueden visualizar o completar desde la aplicación móvil. Esta relación se gestiona mediante la tabla `UserFormulario` y los *endpoints* del conjunto `AsignacionViewSet`, que admiten operaciones masivas para asignar, reemplazar o eliminar formularios asociados a un usuario. De esta manera, la plataforma asegura que cada usuario solo tenga acceso a la información relevante a su rol o tarea dentro del proceso agrícola.

6.4.7. Exportación de datos y compatibilidad

El sistema permite exportar la información recopilada a distintos formatos, principalmente **Excel (XLSX)**, **CSV** y **JSON**. Esta funcionalidad garantiza la interoperabilidad con herramientas analíticas externas y facilita la validación o procesamiento posterior de los datos recolectados. El diseño modular de la API también permite futuras integraciones con sistemas de visualización o plataformas de análisis de datos del sector agrícola, promoviendo la reutilización de la infraestructura desarrollada.

6.4.8. Automatización mediante señales (*signals*)

Para optimizar los flujos de trabajo internos, se implementaron señales (`Django signals`) que automatizan procesos clave, tales como:

- Creación automática de la primera página (“General”) al registrar un nuevo formulario.
- Generación y activación inmediata de una versión inicial del formulario.
- Registro histórico en la tabla de versiones cada vez que se edita un formulario.

- Revocación automática de tokens de acceso cuando se desactiva un usuario o se revoca su permiso de acceso web.

Las funcionalidades del *backend* garantizan la flexibilidad, escalabilidad y seguridad del sistema. Gracias a su arquitectura modular, cada componente (formularios, campos, usuarios y *datasets*) puede evolucionar de forma independiente, lo que permite mantener la coherencia de los datos y la trazabilidad completa de la información recolectada en campo. Esta infraestructura no solo satisface los requerimientos actuales del proyecto agrícola, sino que también establece una base sólida para futuras expansiones y aplicaciones en otros contextos de recopilación y análisis de datos.

6.5. Implementación de la infraestructura *backend*

La implementación de la infraestructura *backend* se llevó a cabo de forma iterativa, siguiendo el marco de trabajo ágil *Scrum*. El desarrollo se organizó en *sprints* semanales, donde cada iteración abordó la construcción y validación de un conjunto específico de funcionalidades. Este enfoque permitió mantener un flujo continuo de trabajo, entregar incrementos funcionales y realizar ajustes oportunos basados en la retroalimentación técnica y funcional.

El proceso inició con la configuración del entorno de desarrollo en Django y la creación de un entorno virtual de Python, lo que permitió aislar las dependencias y asegurar la reproducibilidad del sistema. Posteriormente, se instaló y configuró PostgreSQL como motor de base de datos, seleccionándolo por su estabilidad, cumplimiento de estándares SQL, soporte avanzado para datos relacionales y extensibilidad. La comunicación entre Django y la base de datos se estableció mediante el ORM (*Object Relational Mapper*) nativo del *framework*, lo que permitió definir las entidades directamente en modelos Python sin necesidad de consultas SQL explícitas.

Durante la fase de desarrollo se implementaron los módulos principales del sistema, estructurados de manera modular dentro del proyecto Django, siguiendo las mejores prácticas de separación de responsabilidades y reutilización de código. Los módulos principales creados fueron:

- **formularios**: Encargado de la lógica de creación, edición, duplicación, suspensión y eliminación de formularios.
- **páginas**: Permite la definición de secciones dentro de cada formulario.
- **campos**: Gestiona los diferentes tipos de campos que componen los formularios, con configuraciones personalizables.
- **grupos**: Gestiona campos de clase grupo que permiten subcampos enlazados a ellos.
- ***datasets***: Administra la carga y lectura de archivos externos en formato Excel, utilizados para listas de autocompletado.
- **exportación**: Módulo que centraliza la obtención y descarga de las respuestas almacenadas en la base de datos.

Una de las tareas más relevantes fue la implementación del sistema de versionamiento, diseñado para conservar el historial estructural de los formularios y las páginas. Cada modificación realizada genera una nueva versión sin alterar las anteriores, garantizando la trazabilidad y la compatibilidad con los formularios desplegados en la aplicación móvil.

Los ***endpoints* REST** se implementaron utilizando *Django REST Framework* (DRF). Este *toolkit* permitió definir vistas genéricas y *viewsets* que exponen las operaciones CRUD (crear, leer,

actualizar y eliminar) de forma estandarizada. Los datos intercambiados con los clientes externos se serializan en formato JSON mediante los *serializers*, los cuales incluyen validaciones automáticas para garantizar la consistencia de los datos recibidos y enviados.

Para asegurar la protección de los recursos, se implementó un sistema de autenticación basado en el protocolo OAuth2, mediante la librería *django-oauth-toolkit*. Este componente permite emitir y validar tokens de acceso, asegurando que únicamente usuarios autenticados y con permisos definidos puedan interactuar con la API. Asimismo, se añadieron permisos personalizados que limitan las acciones administrativas a los usuarios autorizados para crear o modificar formularios.

En cuanto al manejo de datos externos, el módulo de ***datasets*** fue diseñado para permitir la carga, validación y lectura de archivos Excel mediante el uso de librerías como *Pandas* y *OpenPyXL*. Estos archivos se transforman en tablas auxiliares dentro de la base de datos PostgreSQL, las cuales pueden ser referenciadas por los campos tipo *dataset* de los formularios. Esto facilita la reutilización de catálogos y listas sin necesidad de definir manualmente los valores.

Posteriormente, se desarrolló el módulo de **exportación de respuestas**, encargado de recuperar los registros provenientes de la aplicación móvil y generar archivos descargables en los formatos Excel (.xlsx), CSV y JSON. Este módulo implementa funciones de limpieza y normalización de datos previas a la exportación, garantizando la integridad y el formato de los archivos generados.

Finalmente, se documentaron todos los ***endpoints*** **REST** mediante la herramienta *Swagger (OpenAPI)*, lo que permite visualizar y probar la API desde una interfaz gráfica. Esto facilita la integración de nuevos desarrolladores o equipos externos, y asegura la trazabilidad de las funcionalidades disponibles.

El resultado de esta implementación fue una infraestructura *backend* completamente funcional, modular y segura, capaz de administrar la estructura de formularios, conectar con la base de datos relacional PostgreSQL y proveer los servicios necesarios para la exportación y análisis de la información recolectada.

6.6. Automatización y despliegue (CI/CD)

Con el fin de asegurar entregables consistentes, reproducibles y trazables, se implementó un proceso de integración y entrega continua (CI/CD) basado en GitHub Actions y Docker Hub. Este proceso automatiza la construcción de la imagen contenedorizada de la API y su publicación en un registro centralizado, habilitando su despliegue en distintos entornos de ejecución sin modificaciones adicionales.

6.6.1. Objetivo del proceso

El objetivo del proceso de CI/CD es estandarizar la generación de artefactos de software (*Docker images*) a partir del código fuente, así como su distribución confiable mediante un registro de contenedores. De este modo, se garantiza la portabilidad del servicio y la reducción de variabilidad entre entornos (desarrollo, pruebas y producción).

6.6.2. Arquitectura del flujo

El flujo se activa ante eventos de *push* sobre las ramas `dev` y `main` o mediante ejecución manual (`workflow_dispatch`). Una vez disparado, el *pipeline*:

1. Recupera el repositorio y configura las herramientas de construcción multi-plataforma (QEMU/Buildx).
2. Autentica de forma segura contra Docker Hub utilizando secretos de GitHub.
3. Construye la imagen de la API y la etiqueta con `latest` y con el hash del *commit* (`SHA`), para asegurar trazabilidad.
4. Publica la imagen en el repositorio `${DOCKERHUB_USERNAME}/santa-ana-api` de Docker Hub.

6.6.3. Configuración del *workflow*

La configuración del *workflow* de GitHub Actions se muestra a continuación. Esta definición describe los disparadores, los pasos de autenticación y la fase de construcción y publicación de la imagen:

```
name: Build & Push Docker (Docker Hub)

on:
  push:
    branches: [ "dev", "main" ]
  workflow_dispatch: {}

jobs:
  docker:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: docker/setup-qemu-action@v3
      - uses: docker/setup-buildx-action@v3

      - name: Login to Docker Hub
        uses: docker/login-action@v3
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}

      - name: Extract metadata
        id: meta
        uses: docker/metadata-action@v5
        with:
          images: ${{ secrets.DOCKERHUB_USERNAME }}/santa-ana-api
          tags: |
            type=raw,value=latest
            type=sha

      - name: Build and push
        uses: docker/build-push-action@v6
        with:
          context: .
          push: true
          platforms: linux/amd64
          tags: ${{ steps.meta.outputs.tags }}
          labels: ${{ steps.meta.outputs.labels }}
```

6.6.4. Seguridad y gestión de secretos

Las credenciales requeridas para la autenticación en Docker Hub se gestionan mediante `GitHub Secrets` (`DOCKERHUB_USERNAME` y `DOCKERHUB_TOKEN`). Este mecanismo evita la exposición de credenciales en el control de versiones y aplica buenas prácticas de *least privilege* y rotación periódica de tokens.

6.6.5. Estrategia de versionado y trazabilidad

El etiquetado dual de imágenes (`latest` y `sha-{commit}`) permite combinar conveniencia operativa y trazabilidad. La etiqueta `latest` facilita la obtención de la versión más reciente en entornos de prueba, mientras que las etiquetas inmutables por *commit* posibilitan despliegues reproducibles y auditorías ex-post en ambientes de producción.

6.6.6. Proceso de despliegue

Una vez publicada, la imagen puede consumirse en cualquier plataforma compatible con contenedores. En este caso, con el fin de hacer uso de la API desplegada, se utilizó la plataforma Render, donde se configuro con la última versión de la imagen de Docker Hub y las variables del .env necesarias para su ejecución.

6.6.7. Portabilidad del despliegue

El enfoque *build once, run anywhere* garantiza independencia de la plataforma de ejecución. La misma imagen publicada en Docker Hub puede desplegarse en servidores *bare-metal*, máquinas virtuales, servicios de contenedores gestionados o clústeres de Kubernetes, sin cambios en el binario ni recompilaciones.

Resultados

Esta sección resume los hallazgos de (i) pruebas funcionales de *endpoints* con Postman, (ii) pruebas de seguridad (autenticación, autorización y acceso a rutas protegidas), y (iii) pruebas de rendimiento y carga con Locust.

7.1. Pruebas funcionales con Postman

7.1.1. Objetivo y diseño de las pruebas

El objetivo de esta sección fue validar de extremo a extremo el flujo *CRUD* de la plataforma (creación, edición y eliminación de formularios, páginas y campos) desde la perspectiva del cliente HTTP. En particular, se buscó confirmar que: (i) los *endpoints* respondan con los códigos de estado correctos en el *happy path* (201 para creaciones, 200 para ediciones parciales y 204 para eliminaciones); (ii) la encadenación de variables (`form_id`, `page_id`, `campo_id`) se conserve entre *requests*; y (iii) el flujo sea idempotente por iteración, es decir, que cada ejecución cree, modifique y elimine sus propios artefactos sin dejar residuos.

Se elaboró una colección de Postman con el siguiente **flujo CRUD**:

1. **Crear formulario** (`POST /api/formularios/`) → guardar `form_id`.
2. **Editar formulario** (`PATCH /api/formularios/{form_id}/`).
3. **Crear página en formulario** (`POST /api/formularios/{form_id}/agregar-pagina/`) → guardar `page_id`.
4. **Editar página** (`PATCH /api/paginas/{page_id}/`).
5. **Crear campo en página** (`POST /api/paginas/{page_id}/campos/`) → guardar `campo_id`.
6. **Editar campo** (`PATCH /api/campos/{campo_id}/`).
7. **Eliminar campo** (`DELETE /api/campos/{campo_id}/`).
8. **Eliminar página** (`DELETE /api/paginas/{page_id}/`).
9. **Eliminar formulario** (`DELETE /api/formularios/{form_id}/`).

El diseño contempla aserciones por paso (código HTTP esperado y persistencia de variables) y una secuencia de limpieza final (eliminaciones) que asegura que cada iteración queda aislada. Se ejecutaron varias iteraciones consecutivas para observar estabilidad como se puede apreciar en la Figura 3, donde cada columna representa los resultados por iteración y cada barra apilada muestra el total de aserciones aprobadas/fallidas por *request*.

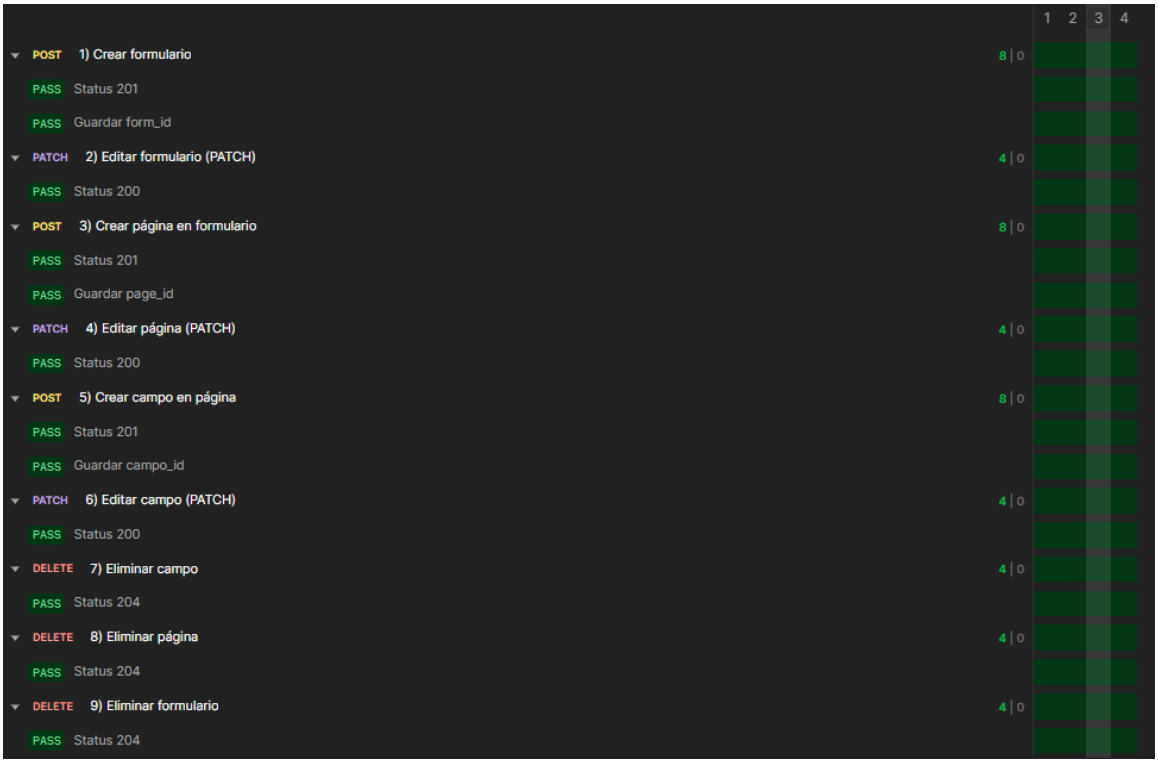


Figura 3: Ejecución de la colección Postman (CRUD *Flow*)

7.1.2. Resultados obtenidos

Todas las aserciones del flujo CRUD se aprobaron en las iteraciones ejecutadas (100 % de *pass rate*). En particular:

- Las **creaciones** devolvieron `201 Created` y almacenaron correctamente los `IDs` (`form_id`, `page_id`, `campo_id`) para los pasos siguientes.
- Las **ediciones** devolvieron `200 OK` confirmando la aplicación de cambios parciales sobre los recursos creados en la misma iteración.
- Las **eliminaciones** devolvieron `204 No Content`, señalando que los recursos fueron removidos exitosamente y evitando efectos colaterales entre iteraciones.

El Cuadro 1 resume el flujo y los códigos esperados/observados por operación (extraídos de los resultados del *runner*).

Cuadro 1: Flujo CRUD validado con Postman y códigos de estado

#	**Operación**	***Endpoint***	**Esperado**	**Observado**
1	Crear formulario	`POST /api/formularios/`	201	201
2	Editar formulario	`PATCH /api/formularios/{form_id}/`	200	200
3	Crear página	`POST /api/formularios/{form_id}/agregar-pagina/`	201	201
4	Editar página	`PATCH /api/paginas/{page_id}/`	200	200
5	Crear campo	`POST /api/paginas/{page_id}/campos/`	201	201
6	Editar campo	`PATCH /api/campos/{campo_id}/`	200	200
7	Eliminar campo	`DELETE /api/campos/{campo_id}/`	204	204
8	Eliminar página	`DELETE /api/paginas/{page_id}/`	204	204
9	Eliminar formulario	`DELETE /api/formularios/{form_id}/`	204	204

Los resultados confirman que la cadena de dependencias entre recursos (formulario → página → campo) se conserva correctamente a lo largo de la iteración: los `IDs` creados se propagan hacia las ediciones y, posteriormente, a las eliminaciones. La simetría del flujo (crear/editar/eliminar) garantiza que no queden residuos, aspecto clave para la higiene de datos en ambientes compartidos de prueba y para la reproducibilidad de corridas automatizadas.

El 100 % de aserciones aprobadas en todas las iteraciones sugiere estabilidad del *happy path* y consistencia en validaciones de capa de negocio (*serializers* y permisos). La Figura 3 nos permite verificar que todas las barras están completamente en verde, sin fallas, y que la distribución de aserciones por *request*/iteración se mantuvo constante.

Desde la perspectiva de mantenibilidad, este flujo es un excelente *smoke test* funcional, pues recorre las rutas principales y verificaría regresiones ante cambios en modelos, validaciones o políticas de permisos. Asimismo, la separación de *setup* (creaciones), *act* (ediciones) y *teardown* (eliminaciones) facilita la integración continua: puede correrse antes del despliegue y fallar temprano si algún eslabón del CRUD se ve afectado.

Con ello la colección de pruebas funcionales en Postman valida satisfactoriamente el flujo CRUD extremo a extremo de la plataforma: todas las operaciones clave responden con los códigos esperados (201/200/204), la encadenación de IDs funciona de forma robusta y el *teardown* elimina los artefactos creados en cada iteración. Esto constituye evidencia de que el *happy path* funcional está correctamente implementado y listo, lo cual puede servir para su ejecución automatizada en la CI/CD, sirviendo como prueba de humo previa a despliegues.

7.2. Pruebas de seguridad

7.2.1. Objetivo y diseño de las pruebas

El objetivo de estas pruebas fue verificar que los controles de autenticación y autorización de la API respondan de forma correcta ante escenarios negativos. En particular, se buscó confirmar que (i) solicitudes de inicio de sesión con credenciales inválidas no emitan tokens de acceso, (ii) usuarios válidos pero sin permisos no obtengan acceso a la plataforma web, y (iii) solicitudes a recursos protegidos sin token o con token inválido sean rechazadas de forma consistente con códigos HTTP 4xx, idealmente 401/403.

Se utilizaron pruebas de carga orientadas a seguridad (escenarios negativos) con Locust, modelando dos clases de usuarios virtuales:

1. **SecurityAuthUser** (autenticación negativa): Ejecuta dos tareas. (a) *Login* con credenciales inexistentes; (b) *Login* con un usuario real sin permiso de acceso web. Ambas deben responder 400/401/403 y, crucialmente, no deben devolver un *access_ token*.

2. **SecurityUser** (acceso a recursos protegidos): Realiza solicitudes a *endpoints* protegidos sin token y con token inválido. La respuesta esperada en ambos casos es 401/403.

Los casos cubren tanto la capa de autenticación (*login*) como la de autorización (verificación de permisos en rutas protegidas), incluyendo rutas críticas y sensibles por su potencial impacto en integridad y confidencialidad como la vista de formularios, y creación de páginas, campos y administración de usuarios.

Se ejecutó una corrida de referencia con **5 usuarios virtuales concurrentes** (*VUs*) durante aproximadamente 1 minuto. Este parámetro de concurrencia replica el escenario operativo esperado en la organización (menos o igual a 5 usuarios web activos de forma simultánea).

Se analizaron las métricas estándar del reporte de Locust:

- ***Requests*** y ***Failures***: número de solicitudes ejecutadas y fallidas (fallas de API).
- **RPS (*Requests Per Second*)**: volumen sostenido de solicitudes por segundo.
- **Tiempos de respuesta**: media, p50 (mediana), p95 y p99.
- **Estadísticas por escenario**: desglose de métricas para cada tipo de petición.

7.2.2. Resultados por escenario

Durante la corrida se procesaron múltiples solicitudes con **0 fallas** a nivel de API. A continuación se muestran las métricas por escenario:

Cuadro 2: Estadísticas por solicitud en pruebas de seguridad negativa

Escenario / Recurso	Req	Fails	Avg (ms)	Min	Max	RPS
Total agregado	227	0	∼436	385	389	3.10
AUTH NEG: usuario válido SIN acceso	128	0	∼446	386	1389	1.75
AUTH NEG: usuario/clave inexistentes	99	0	∼423	385	1295	1.35

Cuadro 3: Percentiles de tiempo de respuesta en pruebas de seguridad negativa

Escenario / Recurso	p50 (ms)	p90 (ms)	p95 (ms)	p99 (ms)
Total agregado	410	440	480	1300
AUTH NEG: usuario válido SIN acceso	410	450	490	1300
AUTH NEG: usuario/clave inexistentes	410	430	470	1300

Cuadro 4: Acceso a rutas protegidas sin token

#	Ruta	Método	Esperado	Observado
1	/api/formularios/	GET	401/403	401/403
2	/api/formularios/	POST	401/403	401/403
3	/api/paginas/	GET	401/403	401/403
4	/api/formularios/bfa564b7-9798-44a8-8a2f-25a8bce5ec4a/agregar-pagina/	POST	401/403	401/403
5	/api/formularios/bfa564b7-9798-44a8-8a2f-25a8bce5ec4a/	PATCH	401/403	401/403
6	/api/formularios/bfa564b7-9798-44a8-8a2f-25a8bce5ec4a/	DELETE	401/403	401/403

Cuadro 5: Acceso a rutas protegidas con token inválido

#	Ruta	Método	Esperado	Observado
1	/api/formularios/	GET	401/403	401/403
2	/api/formularios/	POST	401/403	401/403
3	/api/paginas/	GET	401/403	401/403
4	/api/formularios/bfa564b7-9798-44a8-8a2f-25a8bce5ec4a/agregar-pagina/	POST	401/403	401/403
5	/api/formularios/bfa564b7-9798-44a8-8a2f-25a8bce5ec4a/	PATCH	401/403	401/403
6	/api/formularios/bfa564b7-9798-44a8-8a2f-25a8bce5ec4a/	DELETE	401/403	401/403

Los hallazgos muestran que los mecanismos de seguridad implementados responden adecuadamente ante intentos de acceso no autorizado:

1. **Autenticación negativa**: En ambos casos, *login* con credenciales inexistentes y con usuario sin permiso, el servidor respondió con códigos de error 4xx (400/401/403) y **no emitió tokens**. Esto es central para prevenir escalamiento de privilegios o abuso de credenciales.
2. **Autorización negativa**: Las rutas protegidas rechazaron solicitudes sin token o con token inválido mediante 401/403, indicando que la capa de autorización valida de forma consistente los encabezados de autenticación antes de permitir el acceso a recursos sensibles (creación/edición de formularios, páginas, campos y administración de usuarios).
3. **Rendimiento bajo carga representativa**: Con 5 *VUs* (escenario realista), el sistema mantuvo **p95 $<$ 500 ms** y no presentó fallas. Para pruebas negativas, la latencia observada es adecuada, ya que priorizamos el rechazar acceso por encima de tiempos de respuesta ultrabajos. La presencia de un p99 cercano a 1.3 s sugiere picos ocasionales por contención puntual de recursos, por ejemplo, inicializaciones, verificación de permisos/consultas a BD, *logging* o reintentos internos, sin impacto observable en la corrección de las decisiones de seguridad.

El criterio definido fue: códigos 4xx en *login* inválido (sin token) y 401/403 para accesos no autorizados a recursos protegidos. Las métricas muestran **0 fallas** de API y ausencia de respuestas 2xx en escenarios que debían ser rechazados, por lo que el sistema **cumple** el criterio de seguridad establecido.

Las pruebas de seguridad negativas confirman que la API aplica correctamente los principios de *fail-safe* en autenticación y autorización: no se emiten tokens para credenciales inválidas ni se

permite el acceso a recursos protegidos sin credenciales válidas. En la carga representativa de 5 *VUs*, los percentiles de respuesta se mantuvieron estables (p95 < 500 ms) y no se reportaron fallas de API, lo que valida los controles de acceso para el contexto operativo previsto.

7.3. Pruebas de rendimiento y carga con Locust

7.3.1. Objetivo y diseño de las pruebas

El objetivo fue caracterizar el punto de saturación del *backend* bajo las rutas más críticas por costo de I/O y lógica de negocio, que son la creación, actualización y eliminación de formularios, páginas y campos, los cuales también son los más probables de ejecutarse de forma concurrente por parte de los administradores.

Se modelaron tres escenarios de usuarios concurrentes:

- 70 usuarios: Ejecución sostenida mayor a 5 minutos sin fallos reportados
- 80 usuarios: Inicio de fallos tras llegar a los 4 minutos
- 100 usuarios: Incremento de fallos a partir de los 4 minutos, igualmente

Cada escenario se ejecutó con:

- Conjunto de tareas: POST/PATCH/DELETE de /api/formularios/, /api/paginas/ y /api/campos/.
- Autenticación: Utilizando usuario, contraseña y token bearer generado.

7.3.2. Resultados por escenario

A. 100 usuarios (saturación evidente)

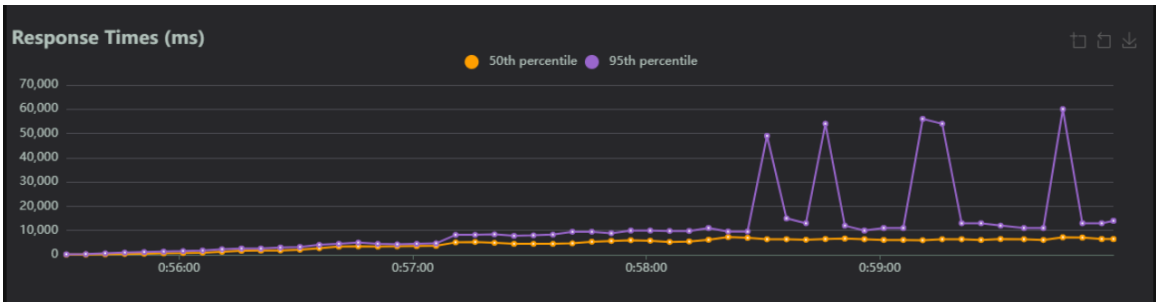


Figura 4: Percentiles de tiempo de respuesta (p50 y p95) con 100 usuarios

Latencia en percentiles altos

La p50 aumenta a valores de segundos, lo que refleja una degradación general en la experiencia típica. La p95 presenta picos muy altos, compatibles con colas largas, esperas por bloqueo en la base de datos y *imeouts* en rutas de escritura (POST/PUT/DELETE). Estos picos suelen concentrarse cuando múltiples operaciones compiten por los mismos recursos.

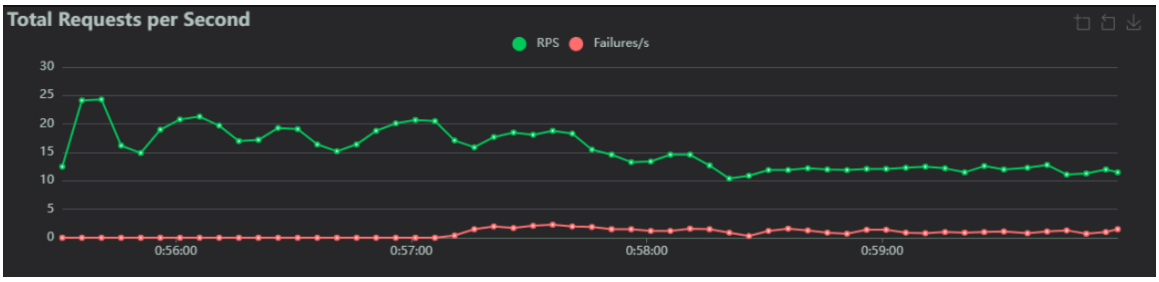


Figura 5: *Requests* por segundo (RPS) y fallos por segundo con 100 usuarios

Relación RPS–Fallos.

Al alcanzar la meseta de 100 usuarios, el sistema muestra fallos recurrentes y un RPS que deja de escalar e incluso disminuye respecto al tramo anterior. Esto indica que la capacidad de proceso está saturada: aunque la concurrencia se mantiene constante, el *throughput* no mejora y la tasa de errores crece.

La aparición de fallos por segundo coincide con el descenso del RPS. Es el patrón esperado cuando la aplicación llega a su techo de capacidad: el servidor o la base de datos empiezan a rechazar o expirar solicitudes, y el *throughput* efectivo se ve limitado por esperas y reintentos.

Causas probables:

- Contención transaccional en tablas de formularios, páginas y campos (bloqueos, conflictos de escritura, índices no óptimos).
- *Timeouts* de la base de datos.
- Autenticación bajo carga que añade latencia y presión a las tablas de sesiones/tokens.

B. 80 usuarios (inicio de saturación)

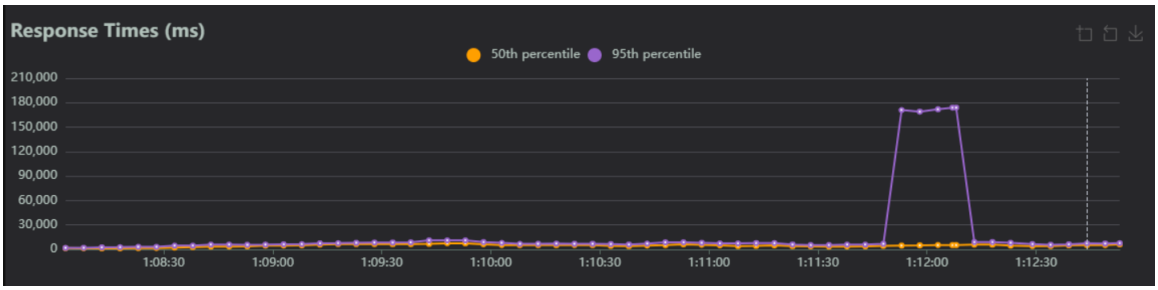


Figura 6: Percentiles de tiempo de respuesta (p50 y p95) con 80 usuarios

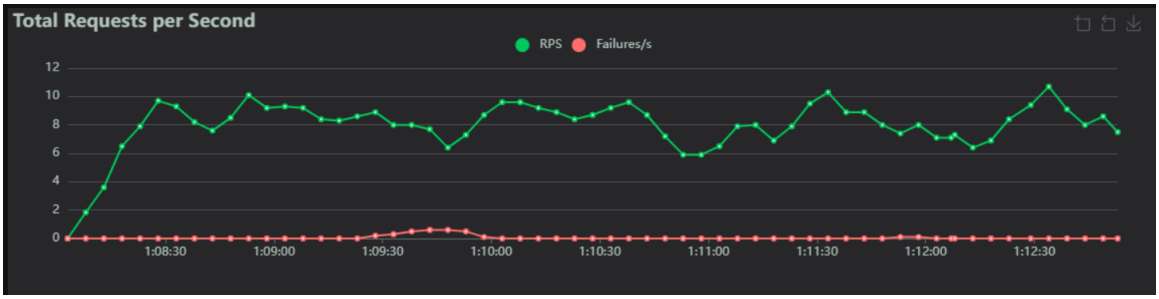


Figura 7: *Requests* por segundo (RPS) y fallos por segundo con 80 usuarios

Comportamiento general

Con 80 usuarios, el sistema mantiene un *throughput* estable (RPS en meseta) y no se observan fallos sostenidos. En la gráfica de percentiles, la mediana (p50) permanece baja y constante, lo que indica que la experiencia típica de CRUD sigue siendo ágil. Se observa, no obstante, un *spike* aislado en p95, que evidencia colas largas transitorias en la cola de peticiones o en la base de datos. Como ese *spike* no se prolonga y no coincide con una caída de RPS ni con un aumento sostenido de fallos, lo interpretamos como un evento puntual.

Este patrón sugiere que 80 usuarios están cerca del punto de inflexión: los percentiles altos comienzan a sensibilizarse, pero no se traducen aún en una degradación sostenida ni en una tasa de error significativa. Es decir, el sistema todavía no se satura, aunque ya asoman señales de presión en la cola de servicio.

C. 70 usuarios

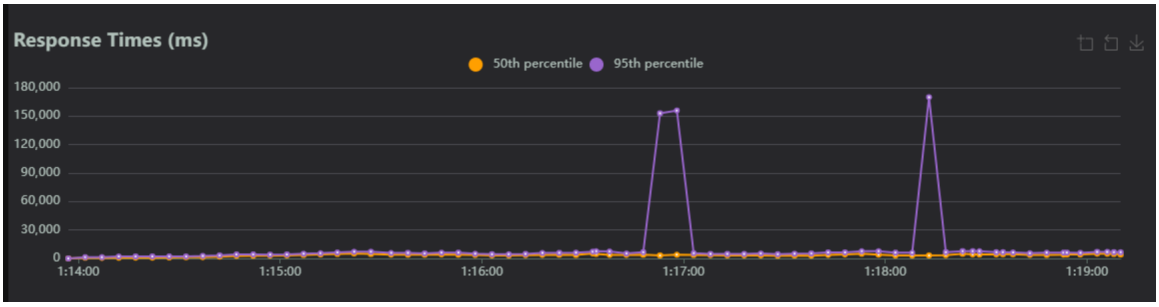


Figura 8: Percentiles de tiempo de respuesta (p50 y p95) con 70 usuarios concurrentes

Latencia: comportamiento de percentiles

La mediana (p50) permanece baja y estable durante la fase sostenida, lo que refleja que la experiencia típica de las operaciones CRUD se mantiene ágil. En la curva de p95 aparecen picos puntuales que no se sostienen en el tiempo ni se traducen en errores. Este patrón es común en pruebas de alta proporción de escrituras (POST/PUT/DELETE), donde pequeñas ventanas de contención en BD, sincronizaciones de disco, *checkpoints* o pausas de GC pueden provocar latencias esporádicas en el percentil alto. La ausencia de fallos simultáneos sugiere que la infraestructura absorbió esos picos sin degradación funcional.

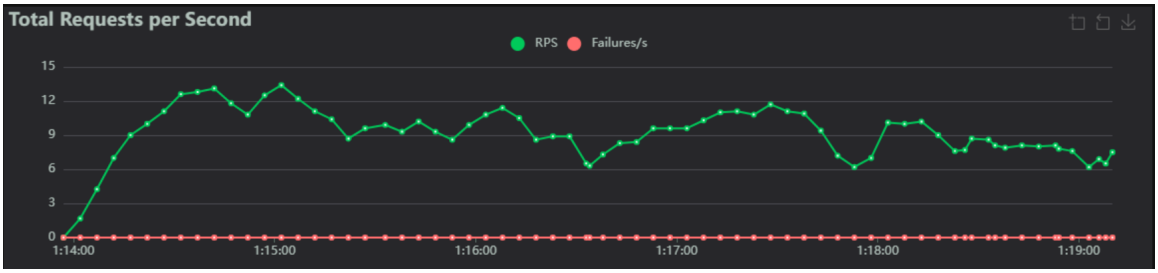


Figura 9: *Requests* por segundo (RPS) con 70 usuarios concurrentes

Rendimiento sostenido (RPS)

El RPS se mantiene dentro de un intervalo estable durante toda la meseta de 70 usuarios. No se observan caídas correlacionadas con los picos de p95, lo que refuerza la idea de que los *outliers* son eventos aislados y breves que no afectan el *throughput* global.

El ingenio utilizará la plataforma con menos de 5 usuarios concurrentes. Dado que, con 70 usuarios, 14 veces ese nivel previsto, el sistema se sostiene sin fallos y con p50 estable, existe un amplio margen de seguridad para la operación real. En otras palabras, el *backend* cumple holgadamente con los requisitos de disponibilidad y rendimiento para la carga esperada.

7.4. Comparación final

La comparación entre la plataforma Digiforms y la API propia desarrollada en este proyecto permite evidenciar las mejoras obtenidas al implementar una infraestructura *backend* personalizada. Mientras que Digiforms representó una solución temporal para la digitalización de formularios agrícolas, la nueva API garantiza independencia tecnológica, acceso directo a los datos y mayor flexibilidad en la administración de formularios y *datasets*. El control total sobre la base de datos, la capacidad de eliminar formularios obsoletos y la exportación estructurada de respuestas representan ventajas significativas en términos de eficiencia, trazabilidad y autonomía operativa. La siguiente tabla resume las principales diferencias entre ambas soluciones.

Cuadro 6: Comparación entre Digiforms y la API propia desarrollada

Criterio	**Digiforms**	**API Propia (*backend* desarrollado)**
Acceso a la base de datos	Los datos se almacenan en servidores externos; el ingenio depende de que Digiforms haga disponible la información.	Acceso directo y completo a la base de datos donde se guardan las respuestas; consulta y análisis bajo demanda.
Disponibilidad de respuestas	Las respuestas se entregan cuando Digiforms las procesa y haga disponible, generando demoras.	Las respuestas quedan disponibles inmediatamente tras el envío desde la app móvil, sin depender de terceros.
Gestión de formularios	Sin eliminación directa; tiende a acumular formularios obsoletos.	Se pueden eliminar o suspender formularios, facilitando la limpieza y organización del sistema.
Estructura y exportación de datos	Exportaciones con estructuras planas o poco intuitivas cuando hay campos anidados.	Exportaciones (Excel/JSON) más ordenadas: los campos detalle se enlazan con su grupo asociado, e identificador de respuesta para lectura intuitiva.
Fuentes de datos para autocompletado	Exige una plantilla de Excel definida para las cargas.	No exige plantilla fija; solo se requiere una columna *id* por fila en la tabla de Excel.

Conclusiones

1. Se desarrolló un *backend* flexible capaz de crear, editar y administrar formularios dinámicos, páginas y campos conforme a las necesidades del administrador, asegurando trazabilidad mediante identificadores únicos y una arquitectura adaptable.

2. Se construyó una base de datos relacional normalizada que almacena de forma íntegra y consistente la estructura de los formularios y la información recopilada, aportando escalabilidad, disponibilidad y trazabilidad al sistema.

3. Se implementó un sistema de procesamiento y validación de datos que permite exportar la información recolectada en formatos estándar como CSV, Excel y JSON, aplicando reglas de limpieza y normalización para asegurar su calidad.

4. Se programó una API REST segura y confiable que sincroniza adecuadamente la aplicación web con la base de datos, validada mediante pruebas de autenticación, autorización y rendimiento, demostrando estabilidad incluso bajo cargas concurrentes superiores a las previstas.

5. Se desarrolló una infraestructura *backend* completa que integra una base de datos relacional, una API REST y mecanismos de exportación interoperables, garantizando el procesamiento correcto y el almacenamiento confiable de los datos agrícolas recolectados en campo, cuyo funcionamiento fue validado mediante pruebas que confirmaron su estabilidad y desempeño frente a formularios adaptables.

6. Las pruebas con Postman arrojaron 100 % de aserciones aprobadas con encadenamiento correcto de IDs, constituyendo un *smoke test* robusto.

7. Las pruebas con Locust confirmaron la correcta denegación de accesos con credenciales inválidas o tokens incorrectos, y el bloqueo consistente de rutas protegidas. Además, el sistema mantuvo un rendimiento estable incluso con 70 usuarios concurrentes sin fallos ni degradación, demostrando un margen de seguridad adecuado.

Recomendaciones

- **Capacitación operativa.** Realizar sesiones breves para personal técnico y de operación sobre el uso de la API, interpretación de códigos HTTP y resolución de incidencias.
- **Reportería de envíos.** Implementar sección de reportería donde se puedan ver todos los formularios enviados y poder hacer una limpieza de los registros de estos formularios para dar más espacio a registros de formularios pendientes de enviar.
- **Integración con fuentes externas.** Además de archivos externos se aconseja implementar el uso de fuentes externas capaz de realizar conexiones a distintas bases de datos SQL para una actualización automática de la información a utilizar para autorrellenados.
- **Seguridad.** Aplicar MFA para cuentas administrativas, implementación de registro automático de terminales en un inicio de sesión y monitoreo de accesos ayudando a mantener la integridad de los datos.
- **Mantenimiento y soporte.** Definir SLOs, rutina de *backups* de la base de datos y planes de mantenimiento y mejora continua garantizando la estabilidad y disponibilidad de la plataforma a largo plazo.

Bibliografía

Ahmed, S. T. (2024). Versatility of Django Framework. https://www.linkedin.com/pulse/versatility-django-framework-syed-taha-ahmed-hcjlf

Amazon Web Services. (s. f.-a). ¿Qué es Django? https://aws.amazon.com/es/what-is/django

Amazon Web Services. (s. f.-b). Front End vs Back End - Difference Between Application Development. https://aws.amazon.com/compare/the-difference-between-frontend-and-backend/

Astera. (2025). ¿Qué es una base de datos? Definición, tipos, beneficios. https://www.astera.com/es/type/blog/what-is-a-database/

Atlassian. (s. f.). Metodología Ágil. https://www.atlassian.com/es/agile

Django REST Framework. (2025). Django REST framework. https://www.django-rest-framework.org/

Django REST Framework. (s. f.). Django REST framework. https://www.django-rest-framework.org/

Django Software Foundation. (2025). Django Project. https://www.djangoproject.com/

Formadores IT. (2023). Postman: qué es y para qué sirve. https://formadoresit.es/que-es-postman-cuales-son-sus-principales-ventajas/

Gabaldón, J. (2021). Metodologías tradicionales vs metodologías ágiles. https://www.linkedin.com/pulse/metodolog%C3%ADas-tradicionales-vs-%C3%A1giles-jos%C3%A9-gabald%C3%B3n

GeeksforGeeks. (2025). ¿Qué es Git? https://www.geeksforgeeks.org/git/what-is-git/

GitLab. (s. f.). Los conceptos básicos del control de versiones. https://about.gitlab.com/es/topics/version-control

Gostev, A., Pykhtin, A., & Popadinets, R. (2019). Selection of Adaptive Agricultural Technologies in Digital Agriculture. *KnE Life Sciences*, *4*(14), 51-61. https://doi.org/10.18502/kls.v4i14.5580

IBM. (2025a). ¿Qué es PostgreSQL? https://www.ibm.com/mx-es/topics/postgresql

IBM. (2025b). API REST. https://www.ibm.com/es-es/think/topics/rest-apis

IBM Engineering Systems Design Rhapsody. (2025). Frameworks and operating systems. https://www.ibm.com/docs/en/engineering-lifecycle-management-suite/design-rhapsody/10.0.1?topic=reference-frameworks-operating-systems

Locust Documentation. (2025). What is Locust? https://docs.locust.io/en/stable/what-is-locust.html

LogicMonitor. (s. f.). What is Azure Blob Storage? https://www.logicmonitor.com/blog/what-is-azure-blob

Mahmood, R. (2023). Django ORM Fundamentals: A Guide to Interacting with Databases. https://www.linkedin.com/pulse/django-orm-fundamentals-guide-interacting-databases-rashid-mahmood

Microsoft Azure. (s. f.). ¿Qué son las bases de datos? https://azure.microsoft.com/es-mx/resources/cloud-computing-dictionary/what-are-databases

Microsoft Learn. (2023). ¿Qué es Scrum? https://learn.microsoft.com/es-es/devops/plan/what-is-scrum

Prytulenets, A. (2025). The Best Backend Frameworks for Speed, Scalability, and Power in 2025. https://5ly.co/blog/best-backend-frameworks/

Red Hat. (2023). ¿Qué es Docker? https://www.redhat.com/es/topics/containers/what-is-docker

Renish, T. W. (2024). Los 11 principales lenguajes de programación backend en 2025. https://webandcrafts.com/blog/backend-languages

Stack Overflow. (2024). Stack Overflow Developer Survey Technology | 2024. https://survey.stackoverflow.co/2024/technology

Stackscale. (2023). 10 sistemas de administración de bases de datos populares. https://www.stackscale.com/es/blog/sistemas-administracion-bases-datos-populares/

Swagger. (s. f.). ¿Qué es Swagger? https://swagger.io/docs/specification/v2_0/what-is-swagger/

UNIR. (2022). Framework: qué es, para qué sirve y algunos ejemplos. https://unirfp.unir.net/revista/ingenieria-y-tecnologia/framework/

Worsley, S. (2024). ¿Qué es Python? Todo lo que necesitas saber para empezar. https://www.datacamp.com/es/blog/all-about-python-the-most-versatile-programming-language

Anexos

11.1. Anexo A. Diagramas técnicos

11.1.1. Modelo entidad–relación

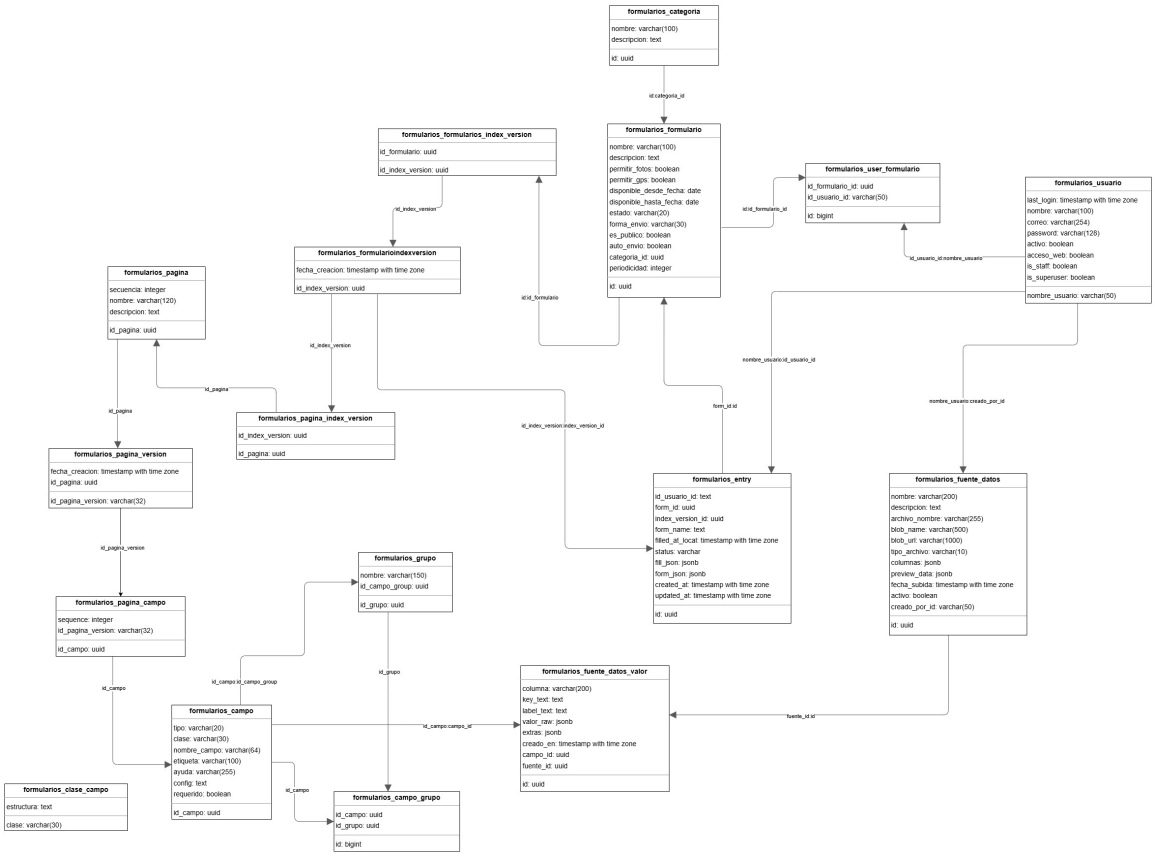


Figura 10: Modelo entidad–relación del *backend* para formularios adaptables

11.1.2. Arquitectura general del sistema

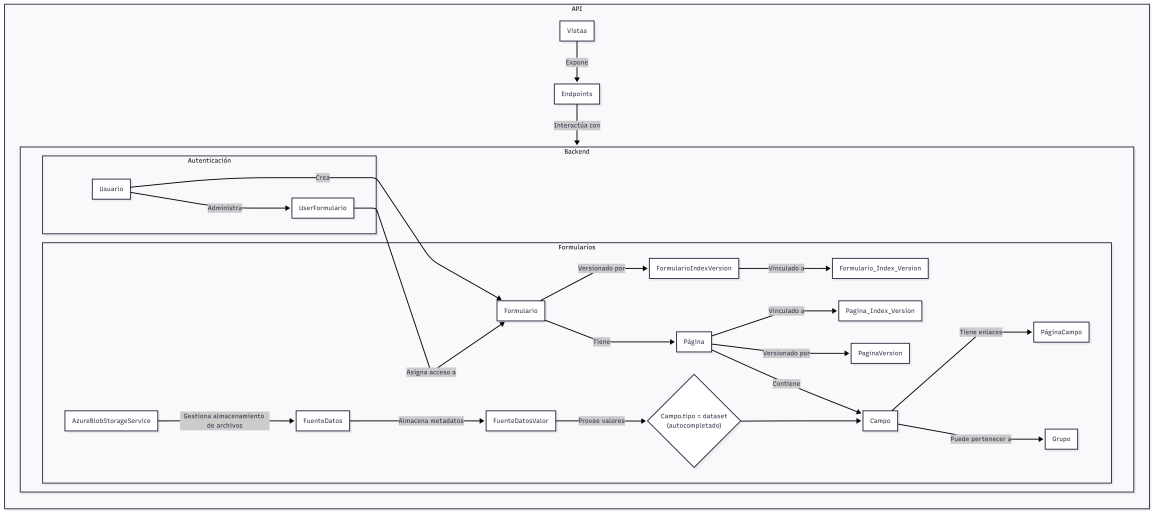


Figura 11: Arquitectura modular del sistema *backend*

11.2. Anexo B. Documentación de la API

11.2.1. Autenticación de usuarios

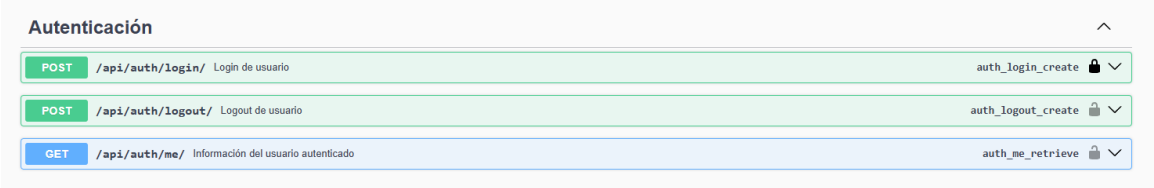


Figura 12: *Endpoints* de *login* y *logout* para el uso de rutas por medio de autenticación de usuarios

11.2.2. Gestión de categorías

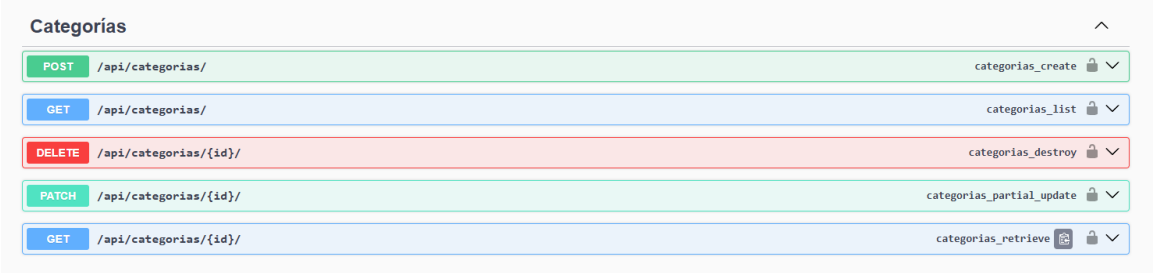


Figura 13: *Endpoints* para la gestión de categorías

11.2.3. Gestión de formularios

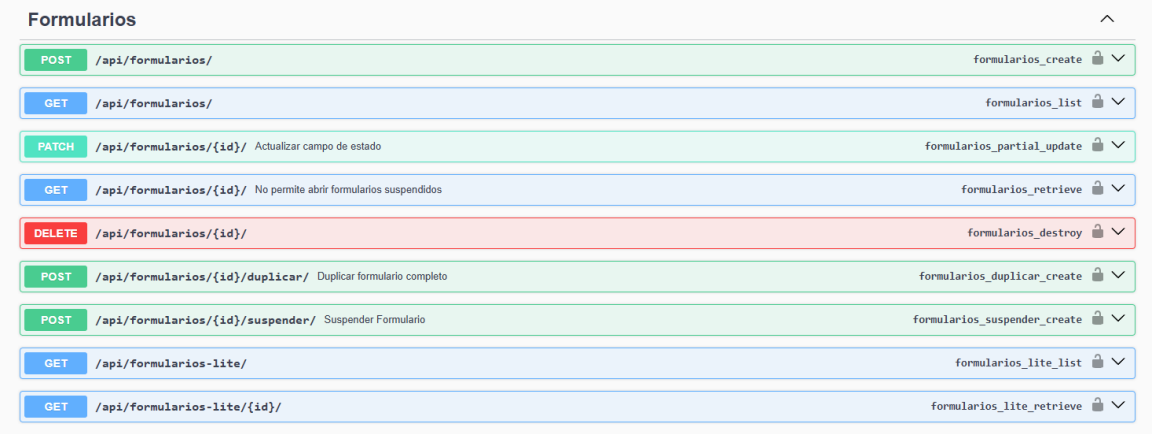


Figura 14: *Endpoints* para la gestión de formularios

11.2.4. Creación de formularios

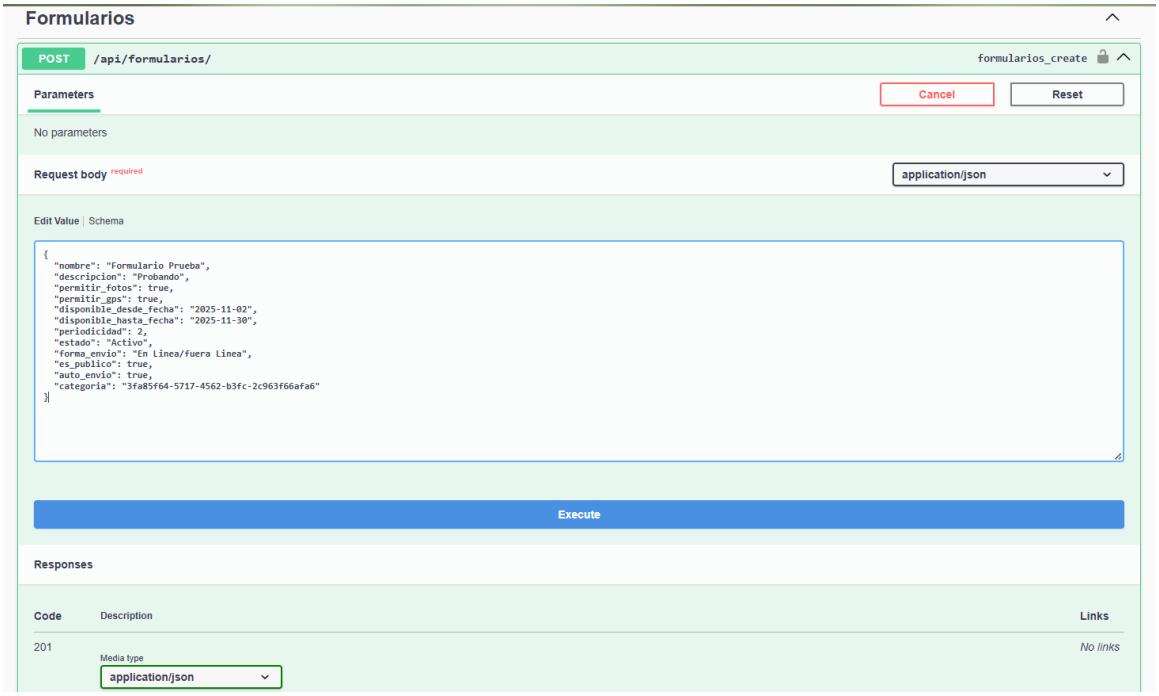


Figura 15: *Endpoint* para la creación de formularios

11.2.5. Gestión de páginas

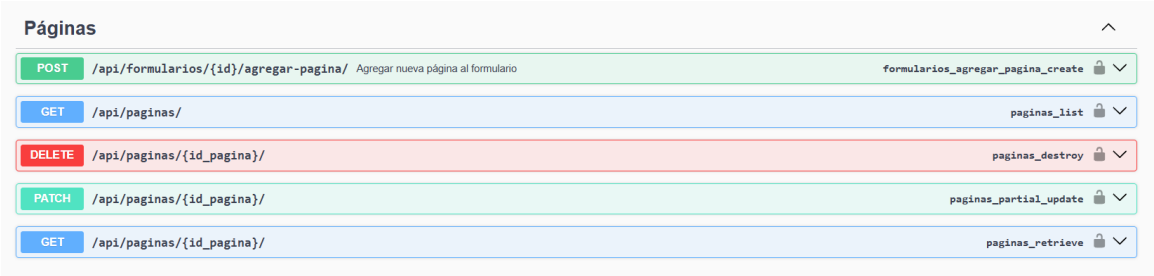


Figura 16: *Endpoints* para la gestión de páginas de formularios

11.2.6. Creación de páginas

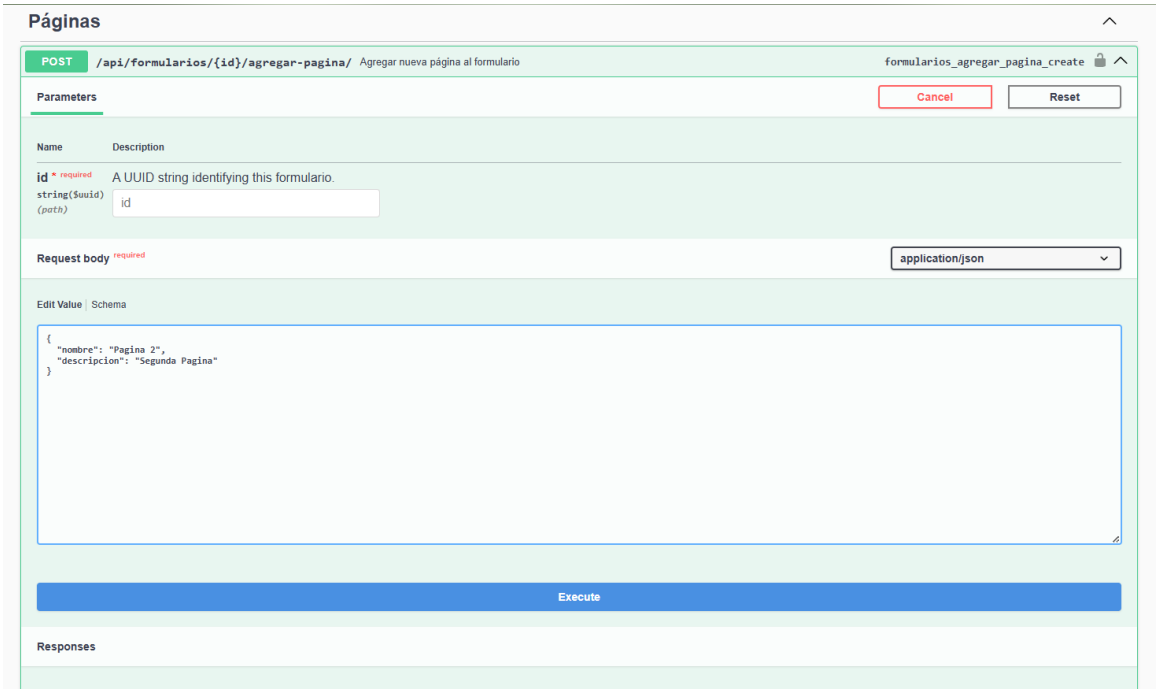


Figura 17: *Endpoint* para la creación de páginas dentro de un formulario

11.2.7. Gestión de campos

Figura 18: *Endpoints* para la gestión de campos

11.2.8. Creación de campos

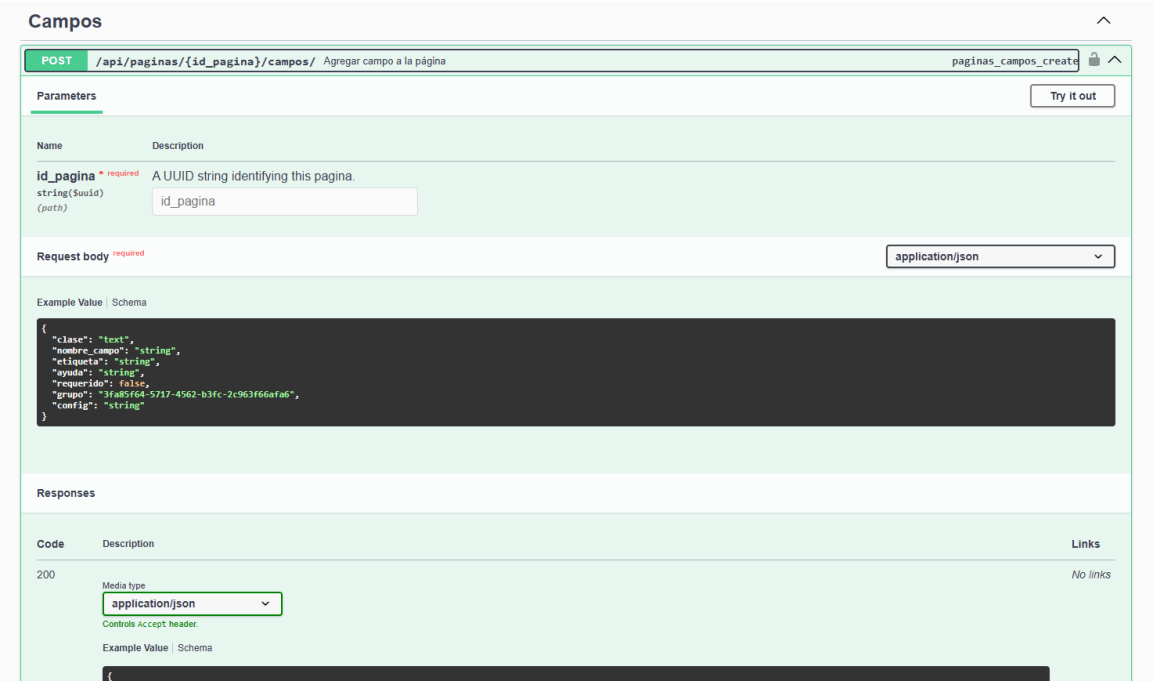


Figura 19: *Endpoints* para la gestión de campos

11.2.9. Gestión de fuentes de datos

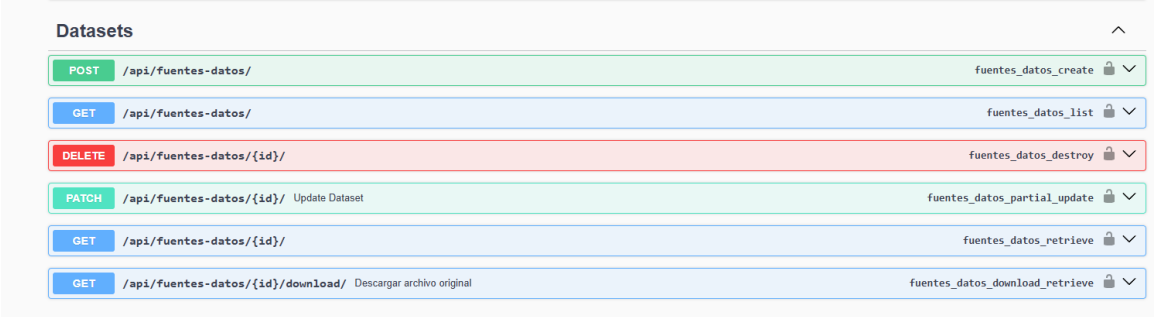


Figura 20: *Endpoints* para la gestión de fuentes de datos

11.2.10. Gestión de usuarios

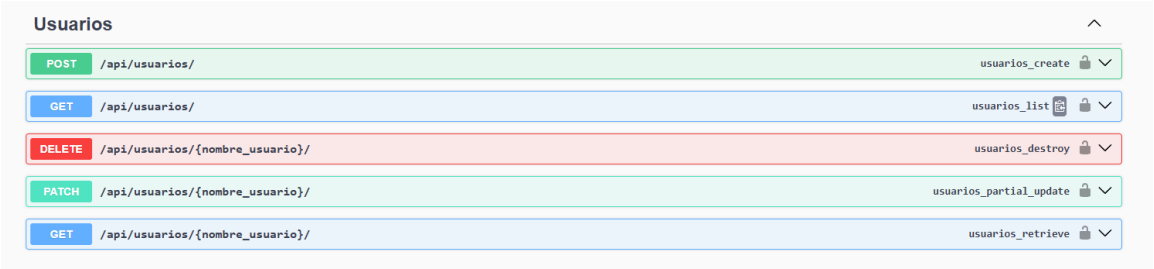


Figura 21: *Endpoints* para la gestión de usuarios

11.2.11. Exportación de datos

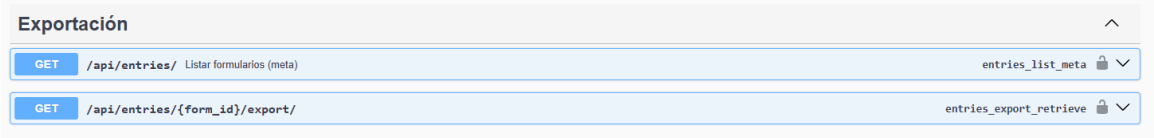


Figura 22: *Endpoints* de exportación de respuestas en formatos Excel, CSV y JSON

Glosario

Backup Copia de seguridad de datos, configuraciones o sistemas, creada periódicamente para prevenir pérdida de información y facilitar su recuperación ante fallos o incidentes.

Dataset Conjunto estructurado de datos organizados para análisis o procesamiento.

Deployment Proceso mediante el cual una aplicación o sistema es puesto en producción.

Endpoint Punto de acceso específico en una API donde se realizan solicitudes.

Frontend Capa de presentación que interactúa directamente con el usuario.

Happy path Escenario ideal donde el sistema sigue el flujo esperado sin errores.

Hash Función criptográfica unidireccional que transforma datos en una cadena de longitud fija.

Load testing Pruebas para evaluar rendimiento del sistema bajo cargas específicas.

Middleware Componente intermedio que facilita la comunicación entre partes de un sistema.

Payload Datos enviados en el cuerpo de una solicitud o respuesta HTTP.

Pipeline Secuencia automatizada de procesos para construcción, prueba o despliegue de software.

Smoke test Prueba inicial rápida que verifica la funcionalidad básica del sistema.

Sprint Periodo fijo en metodologías ágiles donde se completa un conjunto de trabajo.

Throughput Cantidad de trabajo o transacciones procesadas por unidad de tiempo.

Token Cadena utilizada como credencial de autenticación o autorización.

Virtual user Usuario simulado en pruebas de carga para imitar tráfico real.

Workflow Secuencia definida de tareas o procesos, generalmente automatizados.