



Universidad Nacional de
Educación a Distancia

**Máster Universitario de Investigación en
Ingeniería de Software y Sistemas Informáticos**

Código (31105151)

Trabajo de Fin de Máster

***Generación automática de APIs REST
mediante especificaciones
estructuradas y plantillas de código en
Python***

Autor: Diego Cuevas del Valle

Curso: 2025/2026

Tutor: Joan Antonelli

Fecha: octubre de 2026

Resumen

El desarrollo de aplicaciones backend suele implicar la implementación repetitiva de modelos de datos, operaciones CRUD, mecanismos de persistencia, validaciones y pruebas, lo que incrementa el tiempo de desarrollo y la posibilidad de introducir errores. En este contexto, la automatización de tareas mediante técnicas de generación de código constituye una línea de trabajo de gran interés dentro de la Ingeniería del Software.

El presente Trabajo Fin de Máster propone el diseño e implementación de un sistema de generación automática de APIs REST a partir de especificaciones estructuradas definidas en formato JSON. La solución desarrollada permite transformar una descripción de entidades y relaciones en una aplicación backend funcional basada en tecnologías ampliamente utilizadas en el ecosistema Python.

La arquitectura propuesta emplea plantillas Jinja2 para la generación automática de código fuente, integrando FastAPI como framework para la construcción de servicios REST, Pydantic para la validación de datos, SQLAlchemy como capa de acceso a datos y SQLite como sistema de persistencia. Asimismo, el sistema incorpora soporte para la generación automática de operaciones CRUD completas, relaciones entre entidades mediante claves foráneas, navegación relacional, documentación OpenAPI/Swagger, pruebas automatizadas con Pytest y artefactos básicos de despliegue mediante Docker.

Con el fin de evaluar la utilidad de la propuesta, se llevó a cabo un caso de estudio basado en la generación de una API de gestión bibliográfica compuesta por múltiples entidades relacionadas. La evaluación realizada muestra que el sistema permite sustituir la implementación manual de varios componentes repetitivos por la definición de una única especificación JSON. En el caso de estudio se verificó la generación correcta de modelos, operaciones CRUD, persistencia, relaciones entre entidades, pruebas automatizadas y artefactos de despliegue.

Los resultados sugieren que la generación automática basada en especificaciones estructuradas constituye una alternativa viable para acelerar el desarrollo de aplicaciones backend con arquitecturas repetitivas, contribuyendo a mejorar la productividad y la consistencia del software generado. Además, el trabajo identifica diversas líneas de evolución futura orientadas a ampliar las capacidades del sistema mediante nuevas funcionalidades de generación y despliegue.

Palabras clave

Generación automática de código, APIs REST, FastAPI, SQLAlchemy, Ingeniería del Software, Model-Driven Development, Automatización del desarrollo, Backend, Jinja2, Desarrollo dirigido por modelos.

Contenido

1. Introducción	1
1.1 Contexto y motivación.....	1
1.2 Problema abordado	2
1.3 Objetivos del trabajo	3
1.4 Alcance.....	4
1.5 Estructura del documento	5
2. Estado del arte	7
2.1 Desarrollo de APIs REST	7
2.2 Generación automática de código.....	8
2.3 Desarrollo dirigido por modelos (Model-Driven Development).....	9
2.4 Frameworks modernos para desarrollo backend	10
2.5 Herramientas existentes de generación de APIs	12
2.6 Limitaciones de las soluciones actuales.....	14
2.7 Posicionamiento de la propuesta	15
3 Metodología y planificación	16
3.1 Metodología de desarrollo	16
3.2 Planificación del proyecto	17
3.3 Tecnologías utilizadas	19
3.4 Entorno de desarrollo	22
3.5 Estrategia de implementación	24
4. Diseño de la solución	26
4.1 Arquitectura general del sistema.....	26
4.2 Especificación de entrada	28
4.3 Parser de especificaciones	33
4.4 Motor de generación	35
4.5 Arquitectura de código generado.....	40
4.6 Soporte para persistencia.....	45
4.7 Gestión de relaciones	49
4.8 Calidad software.....	53
4.9 Soporte para despliegue.....	56
4.10 Resumen de la arquitectura	58

5. Implementación de la solución	60
5.1 Estructura general del proyecto.....	61
5.2 Implementación del parser.....	62
5.3 Implementación del motor de generación.....	63
5.4 Implementación de las plantillas	64
5.5 Implementación de la persistencia	66
5.6 Implementación de relaciones.....	68
5.7 Implementación de pruebas automáticas	70
5.8 Implementación del soporte Docker	72
5.9 Evolución del prototipo	73
6. Evaluación experimental.....	76
6.1 Objetivos de la evaluación	77
6.2 Caso de estudio.....	77
6.3 Metodología experimental	79
6.4 Resultados obtenidos	81
6.5 Análisis de resultados	84
6.6 Amenazas a la validez	87
6.7 Resumen de la evaluación	89
7. Conclusiones y trabajo futuro	90
7.1 Conclusiones	90
7.2 Aportaciones del trabajo	91
8. Bibliografía	93
9. Lista de siglas y acrónimos	95
10. Anexos.....	96
Anexo A. Especificación JSON utilizada.....	96
Anexo B. Plantillas Jinja2 desarrolladas.....	96
Anexo C. Resultados de las pruebas	97
Anexo D. Dockerfile y dependencias generadas	97

Tabla de Ilustraciones.

Ilustración 1. Visión general del sistema de generación de APIs REST.	24
Ilustración 2. Arquitectura general del sistema.	27
Ilustración 3. Metamodelo de la especificación del dominio.	28
Ilustración 4. Proceso de generación de la aplicación.	37
Ilustración 5. Relación entre la especificación y los artefactos generados.	58
Ilustración 6. Estructura de la aplicación generada.	61
Ilustración 7. Diagrama de clases del caso de estudio.	78

Índice de Tablas

Tabla 1. Fases de desarrollo del proyecto	17
Tabla 2. Resumen de los resultados obtenidos.	82
Tabla 3. Esquema comparativo.	83

1. Introducción

1.1 Contexto y motivación

El desarrollo de aplicaciones software modernas se apoya de forma creciente en arquitecturas basadas en servicios, APIs REST y sistemas backend capaces de exponer funcionalidades de manera estructurada, interoperable y reutilizable. En este contexto, las APIs REST se han consolidado como uno de los mecanismos más utilizados para comunicar aplicaciones, integrar sistemas y construir servicios distribuidos.

Sin embargo, el desarrollo de una API REST funcional suele implicar un conjunto de tareas repetitivas y relativamente estandarizadas. Entre ellas se encuentran la definición de modelos de datos, la creación de operaciones CRUD, la incorporación de validaciones, la configuración de mecanismos de persistencia, la documentación de endpoints y la escritura de pruebas básicas. Aunque estas tareas son necesarias, muchas veces presentan patrones comunes que se repiten entre proyectos y entidades.

Esta repetición puede provocar un aumento del tiempo de desarrollo, inconsistencias entre componentes, errores derivados de implementaciones manuales y dificultades para mantener una estructura homogénea en el código. Además, cuando el número de entidades aumenta, también crece el esfuerzo necesario para mantener modelos, rutas, validaciones y relaciones de forma coherente.

Ante esta situación, la generación automática de código aparece como una alternativa de interés dentro de la Ingeniería del Software. Mediante el uso de especificaciones estructuradas, es posible describir el dominio de una aplicación a un nivel más abstracto y utilizar dicha descripción como entrada para generar automáticamente partes significativas del sistema.

Este Trabajo Fin de Máster se sitúa en este contexto y plantea el diseño e implementación de un prototipo capaz de generar automáticamente APIs REST a partir de especificaciones estructuradas definidas en formato JSON. La motivación principal es estudiar en qué medida este enfoque puede reducir el esfuerzo de desarrollo backend, mejorar la consistencia del código generado y facilitar la construcción de aplicaciones con estructura repetitiva.

1.2 Problema abordado

A pesar de la existencia de numerosos frameworks y herramientas para el desarrollo backend, la construcción de APIs REST continúa requiriendo una cantidad considerable de trabajo manual. En la mayoría de los proyectos es necesario definir modelos de datos, implementar operaciones CRUD, configurar mecanismos de persistencia, establecer relaciones entre entidades, incorporar validaciones y desarrollar pruebas básicas de funcionamiento.

Aunque estas tareas son fundamentales para garantizar el correcto funcionamiento de la aplicación, gran parte de ellas siguen patrones repetitivos que apenas difieren entre proyectos o incluso entre entidades de una misma aplicación. Como consecuencia, los desarrolladores dedican una parte significativa de su tiempo a implementar código de carácter estructural en lugar de centrarse en la lógica específica del dominio.

Esta situación genera diversos inconvenientes. En primer lugar, incrementa el tiempo necesario para desarrollar nuevas aplicaciones o ampliar sistemas existentes. En segundo lugar, aumenta la probabilidad de introducir errores derivados de la implementación manual de componentes similares. Por último, dificulta la homogeneidad y mantenibilidad del código cuando el número de entidades y funcionalidades crece.

El problema abordado en este trabajo consiste en analizar si es posible reducir este esfuerzo mediante un enfoque basado en generación automática de código. Concretamente, se plantea la utilización de especificaciones estructuradas en formato JSON como mecanismo para describir entidades, atributos, validaciones y relaciones, permitiendo posteriormente generar de forma automática una API REST funcional.

La hipótesis subyacente es que una descripción suficientemente expresiva del dominio puede utilizarse para automatizar gran parte del desarrollo backend, reduciendo el trabajo repetitivo y mejorando la productividad sin comprometer la funcionalidad obtenida.

Por tanto, el problema principal que se pretende abordar es la reducción del esfuerzo asociado al desarrollo manual de APIs REST mediante técnicas de generación automática de código basadas en especificaciones estructuradas.

1.3 Objetivos del trabajo

El objetivo principal de este Trabajo Fin de Máster es diseñar, implementar y evaluar un sistema capaz de generar automáticamente APIs REST funcionales a partir de especificaciones estructuradas definidas en formato JSON, reduciendo el esfuerzo asociado al desarrollo manual de aplicaciones backend.

Para alcanzar este objetivo general, se plantean los siguientes objetivos específicos:

O1. Diseñar una especificación estructurada para la definición de APIs

Definir un formato basado en JSON que permita describir entidades, atributos, restricciones de validación y relaciones entre elementos del dominio de forma sencilla y extensible.

O2. Implementar un motor de generación automática

Desarrollar un sistema capaz de interpretar las especificaciones de entrada y generar automáticamente los diferentes componentes que conforman una aplicación backend.

O3. Automatizar la generación de modelos y operaciones CRUD

Generar automáticamente modelos de datos, endpoints REST y operaciones CRUD básicas, minimizando la necesidad de intervención manual por parte del desarrollador.

O4. Incorporar mecanismos de validación y persistencia

Integrar validaciones automáticas mediante Pydantic y mecanismos de persistencia utilizando SQLAlchemy y SQLite, garantizando que las aplicaciones generadas sean completamente funcionales.

O5. Soportar relaciones entre entidades

Permitir la definición y generación automática de relaciones entre entidades, incluyendo claves foráneas, asociaciones ORM y navegación relacional.

O6. Automatizar tareas complementarias del desarrollo

Generar automáticamente elementos auxiliares como pruebas básicas mediante Pytest y artefactos de despliegue como ficheros de dependencias y configuraciones Docker.

O7. Evaluar experimentalmente la solución propuesta

Analizar el impacto de la generación automática comparando el esfuerzo necesario para desarrollar una API de forma manual frente al enfoque basado en generación, utilizando métricas relacionadas con tiempo de desarrollo, cantidad de código producido y automatización conseguida.

O8. Identificar limitaciones y posibles líneas futuras

Analizar las restricciones actuales del prototipo y proponer posibles extensiones que permitan ampliar sus capacidades en futuras investigaciones o desarrollos.

En conjunto, estos objetivos persiguen demostrar la viabilidad de la generación automática de código como mecanismo para incrementar la productividad en el desarrollo de aplicaciones backend basadas en APIs REST.

1.4 Alcance

El presente trabajo se centra en el diseño, implementación y evaluación de un prototipo de generación automática de APIs REST orientado a aplicaciones backend desarrolladas en Python.

El sistema desarrollado permite generar automáticamente distintos componentes software a partir de especificaciones estructuradas definidas en formato JSON. Entre las funcionalidades implementadas se encuentran la generación de modelos de datos, operaciones CRUD, validaciones automáticas, persistencia mediante bases de datos relacionales, relaciones entre entidades, documentación de la API, pruebas básicas automatizadas y artefactos de despliegue.

La solución propuesta utiliza como tecnologías principales FastAPI, Pydantic, SQLAlchemy, SQLite y Jinja2, integradas mediante un proceso de generación basado en plantillas. El resultado obtenido es una aplicación backend funcional capaz de ser ejecutada y validada sin necesidad de implementar manualmente los componentes generados.

La evaluación del sistema se realiza mediante un caso de estudio centrado en una API de gestión bibliográfica, utilizada como escenario experimental para validar el funcionamiento de las distintas capacidades incorporadas al generador y para analizar el impacto de la automatización en términos de productividad.

No obstante, el alcance del trabajo se limita a la generación de aplicaciones backend y no contempla la generación automática de interfaces frontend ni el desarrollo de aplicaciones completas de extremo a extremo. Del mismo modo, aunque se incorporan mecanismos básicos de despliegue mediante Docker, no se aborda la gestión avanzada de infraestructuras, la orquestación de contenedores ni la integración continua.

Asimismo, el sistema no incluye funcionalidades avanzadas como autenticación basada en JWT, control de acceso por roles, generación de migraciones automáticas de bases de datos, soporte para GraphQL o generación de aplicaciones distribuidas. Estas capacidades se consideran posibles líneas de evolución futura, pero quedan fuera del alcance del presente trabajo.

Por tanto, el alcance de esta investigación se centra en demostrar la viabilidad y utilidad de la generación automática de APIs REST funcionales a partir de especificaciones estructuradas, evaluando sus beneficios y limitaciones dentro de un entorno controlado de desarrollo backend.

1.5 Estructura del documento

La presente memoria se organiza en varios capítulos que describen de forma progresiva el problema abordado, la solución desarrollada y los resultados obtenidos.

El Capítulo 1 introduce el contexto del trabajo, la motivación que justifica su realización, el problema abordado, los objetivos perseguidos y el alcance de la propuesta.

El Capítulo 2 presenta el estado del arte relacionado con el desarrollo de APIs REST, la generación automática de código y los enfoques basados en desarrollo dirigido por modelos, analizando las principales soluciones existentes y sus limitaciones.

El Capítulo 3 describe la metodología seguida durante el desarrollo del proyecto, así como la planificación realizada y las tecnologías seleccionadas para la implementación del prototipo.

El Capítulo 4 aborda el diseño de la solución propuesta, detallando la arquitectura general del sistema, la estructura de las especificaciones de entrada y el proceso de generación automática.

El Capítulo 5 presenta la implementación realizada, describiendo los distintos componentes desarrollados, las funcionalidades incorporadas y la evolución del sistema hasta alcanzar la versión final del prototipo.

El Capítulo 6 recoge la evaluación experimental llevada a cabo, incluyendo el caso de estudio utilizado, las métricas consideradas, la comparativa entre desarrollo manual y generación automática y el análisis de los resultados obtenidos.

El Capítulo 7 expone las conclusiones derivadas del trabajo realizado, destacando los principales logros alcanzados y las aportaciones realizadas.

El Capítulo 8 analiza las limitaciones identificadas durante el desarrollo del proyecto y propone posibles líneas de evolución futura.

Finalmente, se incluyen las referencias bibliográficas utilizadas, el listado de siglas y acrónimos empleados y diversos anexos con información complementaria relevante para la comprensión y reproducción del trabajo.

2. Estado del arte

2.1 Desarrollo de APIs REST

Durante las últimas décadas, las arquitecturas basadas en servicios han adquirido una relevancia creciente dentro del desarrollo de software. En este contexto, las Application Programming Interfaces (APIs) se han consolidado como uno de los principales mecanismos para facilitar la comunicación entre aplicaciones, servicios y sistemas distribuidos [5], [6].

Entre los distintos estilos arquitectónicos existentes, REST (*Representational State Transfer*), propuesto por Roy Fielding en el año 2000 [15], se ha convertido en uno de los enfoques más utilizados para el diseño de servicios web. REST define un conjunto de restricciones arquitectónicas orientadas a sistemas distribuidos y establece una forma de interacción basada en recursos y representaciones [15]. Posteriormente, diferentes trabajos han desarrollado y aplicado estos principios al diseño práctico de servicios web REST [16].

La popularización de las arquitecturas orientadas a servicios, las aplicaciones web modernas y los entornos *cloud* ha contribuido a extender el uso de APIs como mecanismo de integración entre sistemas [17], [20].

El desarrollo de una API REST suele implicar la implementación de diversos componentes, entre los que destacan los modelos de datos, los mecanismos de validación, los servicios de persistencia, la documentación técnica y los *endpoints* encargados de gestionar las operaciones CRUD (*Create, Read, Update y Delete*). Aunque la lógica específica de negocio puede variar entre aplicaciones, una parte importante de estos elementos presenta patrones recurrentes, fenómeno habitual dentro del desarrollo software [5], [6].

Esta repetición ha favorecido la utilización de frameworks, abstracciones y componentes reutilizables orientados a reducir el esfuerzo de implementación. La reutilización y separación de responsabilidades constituyen principios ampliamente estudiados dentro de la Ingeniería del Software y del diseño de arquitecturas software [5], [7], [17].

En consecuencia, resulta de interés investigar mecanismos que permitan elevar el nivel de abstracción del desarrollo backend, reduciendo el esfuerzo asociado a tareas repetitivas y favoreciendo la generación automática de componentes software a partir de descripciones de mayor nivel [2], [3], [4].

2.2 Generación automática de código

La generación automática de código constituye una línea consolidada dentro de la Ingeniería del Software, especialmente vinculada a los enfoques de modelado, lenguajes específicos de dominio y desarrollo dirigido por modelos [1], [2], [3], [4].

Su objetivo principal consiste en reducir la intervención manual en determinadas tareas de implementación mediante la producción automática de artefactos software a partir de modelos o especificaciones situados en un nivel superior de abstracción [2], [3].

Tradicionalmente, el desarrollo de aplicaciones requiere que los programadores implementen manualmente código relacionado con estructuras de datos, validaciones, acceso a bases de datos, interfaces de comunicación y otros componentes técnicos. La reutilización y automatización de patrones recurrentes constituyen mecanismos para reducir parte de este esfuerzo [4], [5].

La generación automática de código aborda este problema permitiendo producir determinados componentes a partir de descripciones estructuradas. Entre los beneficios asociados a los enfoques dirigidos por modelos se encuentran la mejora de la productividad, una mayor consistencia entre artefactos y la reducción de determinadas tareas manuales [2], [3], [4].

Existen diferentes estrategias para la generación automática de software. Algunas utilizan modelos, mientras que otras se apoyan en lenguajes específicos de dominio (*Domain-Specific Languages*, DSL) destinados a representar conceptos propios de un área concreta [1], [4]. En todos estos casos, el objetivo consiste en transformar una representación del sistema en uno o varios artefactos software.

El desarrollo dirigido por modelos ha dado lugar a diferentes técnicas y herramientas orientadas a automatizar parcial o totalmente la construcción de sistemas software [2], [3], [19]. Sin embargo, estos enfoques también presentan desafíos relacionados con la expresividad de los modelos, la definición de las transformaciones y la capacidad de adaptación del código generado [2], [19].

En este contexto, la propuesta desarrollada en el presente trabajo se enmarca dentro de las técnicas de generación automática de código orientadas al desarrollo backend. Concretamente, se plantea la utilización de especificaciones estructuradas en formato JSON como mecanismo para describir entidades, restricciones y relaciones, empleando posteriormente plantillas de generación para producir automáticamente aplicaciones REST funcionales.

2.3 Desarrollo dirigido por modelos (Model-Driven Development)

El Desarrollo Dirigido por Modelos (*Model-Driven Development*, MDD) constituye un paradigma de Ingeniería del Software cuyo objetivo principal consiste en incrementar el nivel de abstracción durante el proceso de desarrollo mediante la utilización de modelos como elementos centrales del diseño y la construcción del sistema [2], [3], [19].

Frente a enfoques en los que el código fuente constituye el principal artefacto de construcción, MDD propone utilizar modelos capaces de describir diferentes aspectos de una aplicación. A partir de estos modelos pueden aplicarse transformaciones que produzcan otros modelos o artefactos software ejecutables [2], [3].

Una de las ideas fundamentales del desarrollo dirigido por modelos consiste en separar la descripción conceptual del sistema de determinados detalles concretos de implementación. De esta manera, parte del esfuerzo puede desplazarse desde la programación manual hacia la elaboración de modelos y la definición de transformaciones [2], [4].

Dentro de este ámbito destaca la iniciativa *Model-Driven Architecture* (MDA), promovida por el Object Management Group. MDA plantea diferentes niveles de abstracción y distingue, entre otros conceptos, entre modelos independientes de plataforma (*Platform Independent Models*, PIM) y modelos específicos de plataforma (*Platform Specific Models*, PSM) [18].

Los enfoques dirigidos por modelos se han asociado a beneficios potenciales relacionados con productividad, reutilización, consistencia y automatización del proceso de desarrollo [2], [3], [4]. Sin embargo, su aplicación también presenta importantes desafíos. Entre ellos se encuentran la necesidad de disponer de modelos suficientemente expresivos, la complejidad de las transformaciones y la dificultad para abordar adecuadamente sistemas software de elevada complejidad [19].

France y Rumpe [19], por ejemplo, identifican diferentes retos de investigación relacionados con la aplicación de técnicas dirigidas por modelos a sistemas software complejos. Por tanto, la automatización proporcionada por estos enfoques no elimina la necesidad de diseñar modelos adecuados ni garantiza por sí misma que cualquier dominio pueda ser generado completamente.

La solución propuesta en este trabajo comparte algunos de los principios fundamentales del desarrollo dirigido por modelos. Aunque no utiliza modelos gráficos tradicionales ni sigue estrictamente MDA, emplea una representación abstracta del dominio mediante especificaciones estructuradas en formato JSON.

Estas especificaciones actúan como modelos simplificados a partir de los cuales se generan automáticamente distintos componentes de una aplicación backend. Desde esta perspectiva, el sistema desarrollado puede considerarse una aproximación práctica inspirada en MDD aplicada al desarrollo de APIs REST.

2.4 Frameworks modernos para desarrollo backend

El desarrollo de aplicaciones backend se apoya habitualmente en frameworks que proporcionan abstracciones y componentes reutilizables para gestionar aspectos comunes como peticiones HTTP, rutas, validación, persistencia o estructuración de aplicaciones [5], [6], [17].

Dentro del ecosistema actual pueden encontrarse diferentes alternativas basadas en lenguajes y filosofías de desarrollo distintas, entre ellas Flask, Django, Express.js, Spring Boot y FastAPI.

Flask

Flask es un framework ligero para el desarrollo de aplicaciones web dentro del ecosistema Python. Su diseño proporciona una estructura básica sobre la que el desarrollador puede incorporar diferentes extensiones y componentes en función de las necesidades de cada aplicación [21].

Su flexibilidad permite adoptar diferentes estructuras de proyecto y ampliar sus capacidades mediante extensiones. Esto facilita su utilización tanto en aplicaciones sencillas como en proyectos de mayor complejidad [21].

Django

Django es un framework de desarrollo web basado en Python que proporciona una infraestructura más amplia y estructurada. Entre sus componentes se encuentra una capa ORM para la definición y manipulación de modelos de datos, además de mecanismos para la construcción de vistas y una interfaz administrativa [22].

Esta integración permite disponer de numerosos componentes dentro del propio framework, reduciendo la necesidad de seleccionar e integrar individualmente herramientas externas para determinadas funcionalidades.

Express.js

Express.js es un framework web para Node.js caracterizado por proporcionar una infraestructura minimalista para el desarrollo de aplicaciones y APIs. Entre sus funcionalidades se encuentran mecanismos de enrutamiento, utilidades HTTP y soporte para diferentes motores de plantillas [23].

Su carácter poco prescriptivo proporciona flexibilidad al desarrollador para seleccionar la organización y los componentes adicionales que formarán parte de la aplicación.

Spring Boot

Spring Boot constituye una solución del ecosistema Java orientada a facilitar la creación de aplicaciones basadas en Spring. Su ecosistema contempla aspectos relacionados con desarrollo web, acceso a datos, configuración, seguridad, pruebas, monitorización y despliegue de aplicaciones [24].

Esta amplitud funcional lo convierte en una alternativa especialmente orientada a aplicaciones empresariales y sistemas que requieren una infraestructura de desarrollo más completa.

FastAPI

FastAPI es el framework seleccionado en este trabajo para la generación de APIs REST. Su documentación oficial describe características como el uso de anotaciones de tipos de Python, la integración con mecanismos de validación de datos y la generación automática de documentación basada en OpenAPI [8].

Entre las características relevantes para el presente proyecto se encuentran la utilización del sistema de tipos de Python, la integración con Pydantic para la validación de datos [8], [9], la generación automática de esquemas y documentación OpenAPI [8] y el soporte para el desarrollo de servicios REST [8].

Estas características resultan especialmente adecuadas para un sistema basado en generación automática de código, ya que gran parte de la estructura de las aplicaciones puede expresarse mediante modelos, tipos y rutas generables de forma sistemática.

Justificación de la tecnología seleccionada

Tras estudiar las alternativas consideradas, FastAPI fue seleccionado como framework principal para la implementación del prototipo. La decisión estuvo motivada principalmente por su integración con Pydantic, su utilización de anotaciones de tipos y su soporte para la generación automática de documentación OpenAPI [8], [9].

Además, su arquitectura resulta compatible con la organización modular planteada en este trabajo y con la generación mediante plantillas.

2.5 Herramientas existentes de generación de APIs

La automatización de la construcción de software puede abordarse mediante diferentes mecanismos, entre ellos generadores de código, lenguajes específicos de dominio y transformaciones de modelos [1], [2], [4].

En el ámbito específico de las APIs existen herramientas como OpenAPI Generator y Swagger Codegen, además de generadores de proyectos y plataformas Low-Code y No-Code.

OpenAPI Generator

OpenAPI Generator es una herramienta capaz de generar automáticamente clientes, servidores y documentación a partir de documentos definidos conforme a OpenAPI 2.0 y 3.x [25]. La herramienta dispone de numerosos generadores para diferentes lenguajes y tecnologías y permite personalizar las plantillas utilizadas durante el proceso de generación [25].

Este planteamiento permite utilizar el contrato OpenAPI como elemento de entrada para producir diferentes artefactos relacionados con la implementación y utilización de una API.

Swagger Codegen

Swagger Codegen permite generar automáticamente bibliotecas cliente, *server stubs* y documentación a partir de una descripción OpenAPI [26]. Además, permite personalizar los procesos de generación mediante plantillas y ofrece integración con diferentes flujos de desarrollo [26].

Este enfoque comparte con OpenAPI Generator la utilización de una descripción formal de la interfaz como punto de partida del proceso de generación.

Yeoman

Yeoman es una herramienta de *scaffolding* orientada a automatizar la creación de estructuras iniciales de proyectos. Su funcionamiento se basa en un ecosistema de generadores que pueden crear proyectos completos o partes concretas de estos mediante plantillas y configuraciones predefinidas [27].

A diferencia de las herramientas centradas específicamente en OpenAPI, Yeoman adopta un enfoque más general orientado a la creación de estructuras y configuraciones iniciales para distintos tipos de proyectos.

Plataformas Low-Code y No-Code

Las plataformas Low-Code y No-Code constituyen otra aproximación a la reducción del esfuerzo de programación manual. Estas tecnologías buscan permitir la construcción de aplicaciones utilizando mecanismos de mayor nivel de abstracción y reduciendo la cantidad de código escrito directamente por el desarrollador [28].

La literatura reciente identifica el Low-Code/No-Code como una tendencia que amplía la participación en el desarrollo de software más allá de los perfiles tradicionales de ingeniería y analiza tanto sus beneficios como los desafíos asociados a su adopción [28].

Entre las plataformas comerciales existentes pueden encontrarse soluciones como Mendix, OutSystems y Microsoft Power Apps. Aunque su enfoque difiere del utilizado en este trabajo, comparten el objetivo general de incrementar el nivel de abstracción y reducir determinadas tareas de programación manual.

Comparación con la propuesta desarrollada

Las soluciones analizadas representan distintos niveles y estrategias de automatización. Desde una perspectiva general, estas aproximaciones pueden relacionarse con la tendencia de la Ingeniería del Software a elevar el nivel de abstracción y automatizar transformaciones entre modelos y artefactos ejecutables [1], [2], [3], [4].

OpenAPI Generator y Swagger Codegen parten principalmente de contratos OpenAPI para generar clientes, servidores y documentación [25], [26]. Yeoman, por su parte, utiliza generadores para crear estructuras y componentes de proyectos [27], mientras que las plataformas Low-Code/No-Code proporcionan mecanismos de mayor nivel para reducir la necesidad de programación manual [28].

La propuesta desarrollada en este trabajo parte de una descripción estructurada del dominio mediante entidades, atributos, validaciones y relaciones expresadas en formato JSON.

A partir de dicha descripción, el sistema desarrollado genera automáticamente modelos Pydantic, endpoints CRUD, validaciones, modelos SQLAlchemy, persistencia SQLite, relaciones ORM, pruebas básicas, documentación OpenAPI y artefactos de despliegue.

La diferencia fundamental respecto al enfoque de generación basado exclusivamente en contratos de API reside, por tanto, en el nivel de descripción empleado por la propuesta. El objetivo es representar fundamentalmente elementos del dominio y utilizar posteriormente esa representación como fuente para generar diferentes artefactos de implementación, siguiendo una filosofía próxima a los enfoques dirigidos por modelos [2], [3], [4].

2.6 Limitaciones de las soluciones actuales

Los enfoques de generación automática ofrecen ventajas potenciales, pero también presentan limitaciones. La literatura sobre desarrollo dirigido por modelos señala problemas relacionados con la expresividad de los modelos, la complejidad de las transformaciones y la dificultad de aplicar estos mecanismos a sistemas de elevada complejidad [2], [19].

En los enfoques basados en especificaciones de API, el nivel de automatización depende además de la información disponible en el contrato utilizado como entrada. Herramientas como OpenAPI Generator y Swagger Codegen permiten generar diferentes artefactos a partir de una especificación OpenAPI [25], [26], pero la existencia de dicho contrato no elimina necesariamente todas las decisiones relacionadas con la lógica de negocio o el diseño interno de la aplicación.

De manera similar, los frameworks facilitan numerosas tareas de desarrollo mediante abstracciones y componentes reutilizables, pero su utilización no implica necesariamente la automatización completa de modelos, persistencia, relaciones, pruebas y despliegue [5], [6].

En el caso de las soluciones Low-Code y No-Code, la literatura identifica tanto beneficios como desafíos asociados a su adopción y utilización dentro del desarrollo de software [28]. Por tanto, la reducción de programación manual no elimina necesariamente las decisiones relacionadas con diseño, integración, mantenimiento o evolución de las aplicaciones.

Desde una perspectiva más general, las dificultades identificadas en el ámbito del MDE ponen de manifiesto que existe un compromiso entre el nivel de automatización conseguido y la expresividad, flexibilidad y complejidad de los modelos y transformaciones empleados [2], [19].

En este contexto se sitúa la propuesta presentada en este trabajo, cuyo objetivo es investigar hasta qué punto una especificación estructurada relativamente sencilla puede utilizarse para generar diferentes componentes de una aplicación backend.

2.7 Posicionamiento de la propuesta

Tras analizar los principales enfoques relacionados con desarrollo de APIs REST, generación automática de código y desarrollo dirigido por modelos, la propuesta puede situarse dentro del ámbito de las técnicas de automatización del desarrollo software [2], [3], [4].

La solución comparte características con los generadores de código tradicionales, puesto que produce automáticamente artefactos software a partir de una descripción previa. Asimismo, adopta principios próximos al Desarrollo Dirigido por Modelos, donde una representación abstracta constituye el punto de partida de sucesivas transformaciones [2], [3], [18].

No obstante, el sistema desarrollado no pretende constituir una implementación completa de MDA. La propuesta utiliza una especificación JSON propia como representación simplificada del dominio y emplea plantillas Jinja2 para generar una implementación concreta en Python.

Este planteamiento guarda también relación con los lenguajes específicos de dominio, en la medida en que se define un vocabulario limitado para expresar conceptos concretos —entidades, atributos, validaciones y relaciones— y posteriormente transformar esas descripciones en código [1], [4].

La propuesta desarrollada produce automáticamente modelos de datos, operaciones CRUD, validaciones, persistencia, relaciones ORM, documentación automática y pruebas básicas. Estas características pertenecen al sistema construido en este trabajo y, por tanto, su justificación se realiza posteriormente mediante la descripción de la implementación y su evaluación, y no mediante referencias bibliográficas externas.

Desde una perspectiva académica, la propuesta puede entenderse como una aplicación práctica de principios de generación de código y desarrollo dirigido por modelos al ámbito específico de las APIs REST [2], [3], [4].

Su contribución consiste en diseñar e implementar un prototipo que permita estudiar si una especificación estructurada relativamente sencilla puede automatizar una parte significativa de la construcción de una aplicación backend.

De este modo, el trabajo combina conceptos procedentes de la Ingeniería Dirigida por Modelos, los lenguajes específicos de dominio y las arquitecturas basadas en APIs REST [1], [2], [3], [15], utilizando estas ideas como fundamento para la solución que se describe y evalúa en los capítulos posteriores.

3 Metodología y planificación

3.1 Metodología de desarrollo

El desarrollo del presente Trabajo Fin de Máster se llevó a cabo siguiendo una metodología iterativa e incremental. Esta decisión se adoptó debido al carácter exploratorio del proyecto, cuyo objetivo principal consistía en diseñar y validar un prototipo de generación automática de APIs REST a partir de especificaciones estructuradas.

A diferencia de metodologías secuenciales tradicionales, donde todas las fases se planifican y ejecutan de forma estrictamente lineal, el enfoque iterativo permite construir progresivamente la solución, incorporando nuevas funcionalidades y refinando componentes existentes conforme se adquiere un mayor conocimiento del problema y de las tecnologías empleadas.

El proceso de desarrollo se estructuró en varias iteraciones sucesivas, cada una de ellas orientada a la incorporación de nuevas capacidades dentro del sistema. En cada fase se implementó una funcionalidad concreta, se validó su correcto funcionamiento mediante pruebas prácticas y posteriormente se utilizó como base para la siguiente iteración.

La primera etapa se centró en la definición del formato de especificación basado en JSON y en la construcción del mecanismo encargado de interpretar dichas especificaciones. Posteriormente se desarrolló el motor de generación automática utilizando plantillas Jinja2, permitiendo producir modelos de datos y componentes básicos de una API REST.

Una vez establecida la capacidad básica de generación, se incorporaron progresivamente nuevas funcionalidades, incluyendo validaciones automáticas mediante Pydantic, generación de operaciones CRUD, integración con SQLite y SQLAlchemy, soporte para relaciones entre entidades y generación automática de pruebas.

Finalmente, se añadieron mecanismos complementarios orientados a mejorar la calidad y capacidad de despliegue del sistema, como la generación automática de pruebas mediante Pytest y la creación de artefactos Docker para facilitar la distribución de las aplicaciones generadas.

Este enfoque incremental permitió validar continuamente los avances obtenidos, detectar problemas de forma temprana y adaptar la arquitectura conforme evolucionaban los requisitos del proyecto. Como resultado, el sistema fue creciendo progresivamente desde un generador básico de modelos hasta una solución capaz de producir aplicaciones backend funcionales y completas.

La metodología seguida resulta especialmente adecuada para proyectos de investigación aplicada, donde la experimentación y la evolución gradual de la solución constituyen elementos fundamentales para alcanzar los objetivos planteados.

3.2 Planificación del proyecto

El desarrollo del proyecto se organizó en diferentes fases de trabajo, cada una de ellas orientada a la incorporación progresiva de nuevas capacidades dentro del sistema de generación automática. Esta planificación permitió evolucionar gradualmente desde un prototipo básico hasta una solución capaz de generar aplicaciones backend funcionales y completas.

Tabla 1. Fases de desarrollo del proyecto

Fase	Objetivo principal
Fase 1	Análisis del problema y estudio de tecnologías
Fase 2	Diseño de la especificación JSON
Fase 3	Implementación del parser de especificaciones
Fase 4	Desarrollo del motor de generación mediante Jinja2
Fase 5	Generación automática de modelos Pydantic
Fase 6	Generación automática de routers FastAPI
Fase 7	Incorporación de validaciones automáticas
Fase 8	Integración de persistencia mediante SQLite
Fase 9	Integración de SQLAlchemy ORM
Fase 10	Implementación de CRUD persistente
Fase 11	Incorporación de relaciones entre entidades
Fase 12	Generación automática de relaciones ORM
Fase 13	Implementación de respuestas relacionales
Fase 14	Generación automática de pruebas con Pytest
Fase 15	Generación automática de artefactos Docker
Fase 16	Evaluación experimental y validación

Durante las primeras fases se abordó el diseño conceptual de la solución, definiendo la estructura de las especificaciones de entrada y la arquitectura general del sistema de generación.

Posteriormente, se desarrollaron los mecanismos necesarios para interpretar las especificaciones y producir automáticamente los distintos componentes software requeridos por una API REST. En esta etapa se incorporaron progresivamente modelos de datos, validaciones, operaciones CRUD y persistencia.

Una vez obtenida una versión funcional del generador, se añadieron características avanzadas como relaciones entre entidades, navegación relacional y generación automática de pruebas, con el objetivo de aproximar el sistema a escenarios de desarrollo reales.

Finalmente, se incorporaron funcionalidades relacionadas con el despliegue y se llevó a cabo una evaluación experimental orientada a analizar el impacto de la generación automática en términos de productividad y reducción del esfuerzo de desarrollo.

La planificación seguida permitió validar cada componente de forma independiente antes de integrarlo en el sistema global, reduciendo riesgos técnicos y facilitando la evolución controlada del prototipo.

3.3 Tecnologías utilizadas

La implementación del sistema desarrollado en este trabajo se apoyó en un conjunto de tecnologías de código abierto ampliamente utilizadas en el ámbito del desarrollo backend moderno. La selección de cada herramienta se realizó considerando criterios relacionados con la simplicidad de integración, la productividad, la documentación disponible y la adecuación a los objetivos del proyecto.

Python

Python fue seleccionado como lenguaje principal de desarrollo debido a su simplicidad sintáctica, amplia adopción en la comunidad científica y tecnológica, y la gran cantidad de bibliotecas disponibles para automatización, desarrollo web y generación de código.

Además, Python facilita la implementación rápida de prototipos y resulta especialmente adecuado para proyectos orientados a la automatización de tareas y la construcción de herramientas de generación.

FastAPI

FastAPI constituye el framework principal utilizado para la generación automática de APIs REST.

Su elección se fundamenta en diversas características relevantes:

- Alto rendimiento de ejecución.
- Soporte nativo para tipado estático.
- Integración directa con Pydantic.
- Generación automática de documentación OpenAPI.
- Facilidad de desarrollo y mantenimiento.

Estas características permiten construir aplicaciones backend modernas minimizando la complejidad de implementación.

Pydantic

Pydantic se utilizó como mecanismo de validación y definición de modelos de datos.

Esta biblioteca permite definir esquemas mediante anotaciones de tipos de Python y realizar validaciones automáticas sobre los datos recibidos por la API.

Gracias a esta integración, las aplicaciones generadas incorporan mecanismos automáticos de validación sin necesidad de implementar lógica adicional.

SQLAlchemy

SQLAlchemy fue empleado como sistema ORM (Object Relational Mapping) para gestionar la persistencia de datos.

Su utilización proporciona una capa de abstracción entre la aplicación y la base de datos, permitiendo trabajar con objetos Python en lugar de consultas SQL directas.

Además, SQLAlchemy ofrece soporte para relaciones entre entidades, claves foráneas y múltiples motores de bases de datos.

SQLite

SQLite fue seleccionado como motor de persistencia principal debido a su simplicidad de configuración y despliegue.

Al tratarse de una base de datos embebida, no requiere procesos adicionales ni servidores externos, lo que facilita considerablemente el desarrollo y la validación experimental del prototipo.

Aunque existen soluciones más avanzadas para entornos empresariales, SQLite resulta suficiente para los objetivos perseguidos en este trabajo.

Jinja2

Jinja2 constituye el núcleo del sistema de generación automática desarrollado.

Esta biblioteca permite definir plantillas parametrizadas capaces de producir código fuente dinámicamente a partir de especificaciones estructuradas.

Mediante el uso de Jinja2 fue posible automatizar la generación de:

- modelos Pydantic;
- routers FastAPI;
- modelos SQLAlchemy;
- pruebas automáticas;
- configuraciones Docker;
- componentes auxiliares.

Pytest

Pytest fue utilizado para la generación y ejecución automática de pruebas básicas sobre las APIs producidas por el sistema.

La incorporación de pruebas automatizadas permite verificar el correcto funcionamiento de los endpoints generados y contribuye a mejorar la calidad del software obtenido.

Docker

Docker se incorporó como mecanismo para facilitar la distribución y despliegue de las aplicaciones generadas.

Aunque la validación experimental se realizó principalmente en un entorno local, el sistema es capaz de producir automáticamente los artefactos necesarios para construir imágenes contenedorizadas, favoreciendo la reproducibilidad y portabilidad de las soluciones generadas.

Visual Studio Code

Visual Studio Code fue utilizado como entorno principal de desarrollo debido a su integración con Python, soporte para depuración, gestión de extensiones y compatibilidad con herramientas de control de versiones.

Síntesis tecnológica

La combinación de estas tecnologías permitió construir una solución completamente basada en software libre y ampliamente respaldada por la comunidad de desarrollo.

Asimismo, todas las herramientas seleccionadas presentan una elevada compatibilidad entre sí, facilitando la integración de los distintos componentes necesarios para implementar el sistema de generación automática propuesto.

3.4 Entorno de desarrollo

El desarrollo y validación del prototipo se llevaron a cabo utilizando un entorno de desarrollo basado en tecnologías ampliamente disponibles y de fácil acceso, con el objetivo de garantizar la reproducibilidad de los resultados obtenidos.

Hardware

El proyecto fue desarrollado sobre un ordenador personal de propósito general, utilizando recursos hardware estándar suficientes para la ejecución de herramientas de desarrollo, servidores locales y bases de datos ligeras.

Dado que la solución propuesta se centra principalmente en la automatización de generación de código y no en tareas de computación intensiva, no fue necesario disponer de equipamiento especializado.

Sistema operativo

La implementación y las pruebas se realizaron sobre un entorno Microsoft Windows 11, utilizando las herramientas y configuraciones habituales para el desarrollo de aplicaciones Python.

La elección de este sistema operativo no afecta a la arquitectura de la solución, ya que todas las tecnologías utilizadas son multiplataformas y pueden ejecutarse igualmente en sistemas Linux o macOS.

Entorno de programación

Como entorno principal de desarrollo se utilizó Visual Studio Code (VS Code), debido a su amplia adopción dentro de la comunidad de desarrollo software y a su integración con las herramientas empleadas en el proyecto.

Entre las funcionalidades utilizadas destacan:

- edición avanzada de código;
- integración con Python;
- terminal integrada;
- depuración;
- gestión de extensiones;
- control de versiones.

Entorno de ejecución

La ejecución del prototipo se realizó utilizando Python 3.13, versión que proporciona soporte para las características modernas del lenguaje empleadas durante el desarrollo.

Los servicios REST generados fueron ejecutados mediante Uvicorn, servidor ASGI recomendado para aplicaciones FastAPI.

Herramientas de validación

La validación funcional de las APIs generadas se realizó principalmente mediante:

- Swagger UI generado automáticamente por FastAPI;
- pruebas manuales sobre endpoints REST;
- pruebas automatizadas mediante Pytest.

Estas herramientas permitieron verificar el correcto funcionamiento de los componentes generados a lo largo de las diferentes fases del proyecto.

Persistencia y almacenamiento

Las pruebas de persistencia se realizaron utilizando SQLite como sistema gestor de bases de datos.

La elección de SQLite permitió simplificar la configuración del entorno experimental y facilitar la ejecución de las aplicaciones generadas sin necesidad de desplegar servidores de bases de datos externos.

Reproducibilidad

Con el objetivo de favorecer la portabilidad y reproducibilidad del sistema, se incorporó soporte para la generación automática de artefactos Docker y ficheros de dependencias, permitiendo reconstruir el entorno de ejecución de forma sencilla en diferentes plataformas.

De este modo, cualquier usuario puede generar nuevamente las aplicaciones desarrolladas a partir de las especificaciones de entrada y ejecutar el resultado utilizando las mismas tecnologías empleadas durante el proyecto.

3.5 Estrategia de implementación

La implementación del sistema se abordó siguiendo una estrategia incremental orientada a la construcción progresiva de un generador automático de aplicaciones backend. Esta estrategia permitió validar cada componente de forma independiente antes de integrarlo dentro de la arquitectura global del sistema.

El punto de partida consistió en definir una representación estructurada del dominio utilizando especificaciones JSON. Estas especificaciones actúan como entrada del sistema y contienen la información necesaria para describir entidades, atributos, restricciones de validación y relaciones entre elementos del modelo de datos.

A partir de dichas especificaciones se desarrolló un mecanismo de análisis encargado de interpretar la información proporcionada y transformarla en una estructura interna que pudiera ser utilizada por el motor de generación.

Una vez procesadas las especificaciones, la estrategia de implementación se basó en la utilización de plantillas parametrizadas desarrolladas mediante Jinja2. Estas plantillas permiten transformar la representación abstracta del dominio en distintos artefactos software necesarios para construir una aplicación backend funcional.

El proceso de generación se organizó en varias capas especializadas. Cada una de ellas se encarga de producir un tipo concreto de componente, incluyendo modelos de datos, routers REST, mecanismos de persistencia, relaciones ORM, pruebas automatizadas y configuraciones auxiliares.

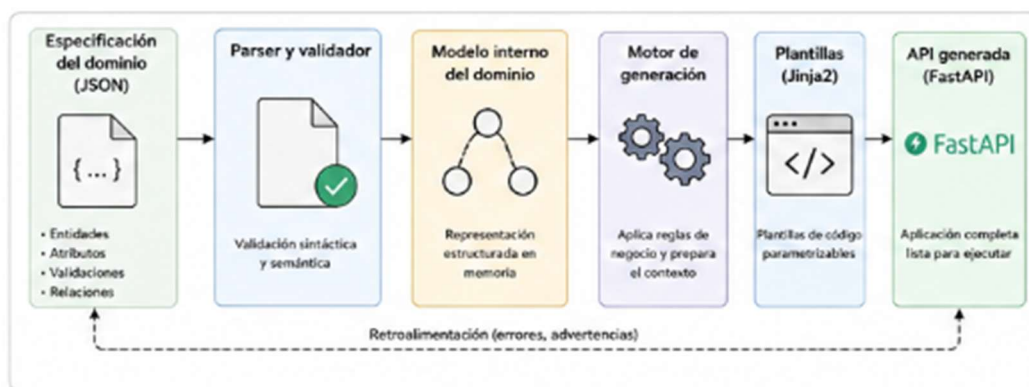


Ilustración 1. Visión general del sistema de generación de APIs REST.

La Figura muestra de forma simplificada el flujo general seguido por el sistema.

Especificación JSON

Parser -> Modelo interno -> Motor de generación (Jinja2) -> Código generado ->

Aplicación FastAPI funcional

Esta arquitectura presenta varias ventajas. En primer lugar, facilita la separación entre la definición del dominio y la implementación concreta de la aplicación. En segundo lugar, permite ampliar el sistema mediante nuevas plantillas sin necesidad de modificar significativamente el núcleo del generador. Finalmente, favorece la reutilización de componentes y simplifica la incorporación de nuevas funcionalidades.

La estrategia seguida permitió evolucionar progresivamente desde un generador básico de modelos hasta una solución capaz de producir aplicaciones backend completas con persistencia, relaciones entre entidades, documentación automática, pruebas y mecanismos básicos de despliegue.

Este enfoque constituye la base arquitectónica sobre la que se construye el sistema presentado en los capítulos siguientes.

4. Diseño de la solución

4.1 Arquitectura general del sistema

La solución desarrollada en este trabajo se basa en una arquitectura orientada a la generación automática de aplicaciones backend a partir de especificaciones estructuradas definidas por el usuario.

El objetivo principal de esta arquitectura consiste en transformar una descripción abstracta del dominio de aplicación en un conjunto de componentes software completamente funcionales, minimizando la necesidad de implementación manual.

La arquitectura propuesta está compuesta por varios módulos independientes que colaboran durante el proceso de generación. Cada uno de ellos es responsable de una tarea específica dentro del flujo general del sistema.

De forma simplificada, el proceso comienza con la definición de una especificación estructurada en formato JSON. Esta especificación describe las entidades del dominio, sus atributos, las restricciones de validación asociadas y las posibles relaciones existentes entre ellas.

Posteriormente, la especificación es procesada por un módulo de análisis encargado de interpretar la información contenida en el documento y transformarla en una representación interna adecuada para el proceso de generación.

Una vez construida dicha representación, el motor de generación utiliza un conjunto de plantillas Jinja2 para producir automáticamente los distintos componentes software necesarios para construir la aplicación backend.

Entre los artefactos generados se encuentran:

- modelos Pydantic;
- routers FastAPI;
- modelos SQLAlchemy;
- configuración de persistencia;
- pruebas automatizadas;
- artefactos de despliegue.

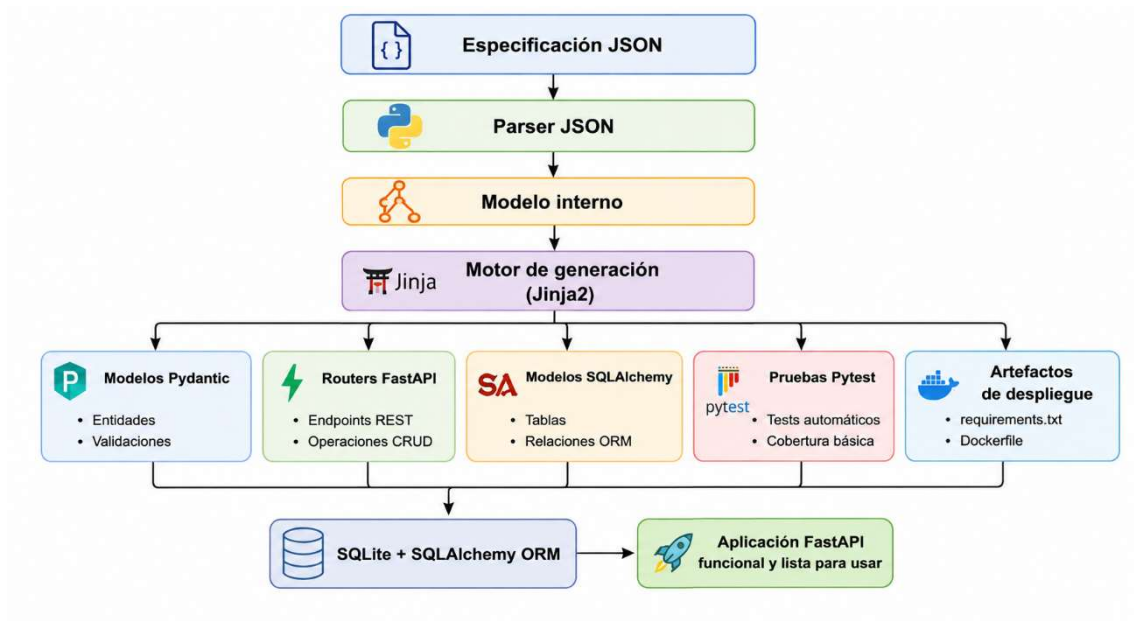


Ilustración 2. Arquitectura general del sistema.

Esta arquitectura presenta diversas ventajas. En primer lugar, favorece la separación de responsabilidades entre los distintos componentes del sistema. En segundo lugar, permite ampliar fácilmente el conjunto de artefactos generados mediante la incorporación de nuevas plantillas. Finalmente, facilita la reutilización de código y la automatización de tareas repetitivas asociadas al desarrollo backend.

Gracias a esta estructura modular, el sistema puede evolucionar progresivamente incorporando nuevas capacidades sin modificar significativamente los componentes ya implementados.

4.2 Especificación de entrada

El funcionamiento del sistema desarrollado se basa en la utilización de especificaciones estructuradas que describen las características de la aplicación a generar. Estas especificaciones constituyen el principal mecanismo de comunicación entre el usuario y el generador, permitiendo definir de forma declarativa los elementos que formarán parte de la API resultante.

La información se representa mediante documentos JSON diseñados específicamente para este trabajo. Dichos documentos contienen la descripción del dominio de aplicación, incluyendo entidades, atributos, restricciones de validación y relaciones entre elementos.

El objetivo de esta aproximación consiste en elevar el nivel de abstracción del proceso de desarrollo. En lugar de implementar directamente modelos, rutas o mecanismos de persistencia, el usuario describe únicamente las características del dominio, delegando en el generador la producción automática del código necesario.

La estructura de la especificación ha sido diseñada para mantener un equilibrio entre simplicidad y expresividad. Por un lado, permite describir aplicaciones de forma sencilla mediante un formato fácilmente interpretable. Por otro, proporciona información suficiente para generar componentes backend funcionales y coherentes.

En los siguientes apartados se describe con detalle la estructura del formato utilizado, así como los mecanismos empleados para representar entidades, validaciones y relaciones dentro del sistema.

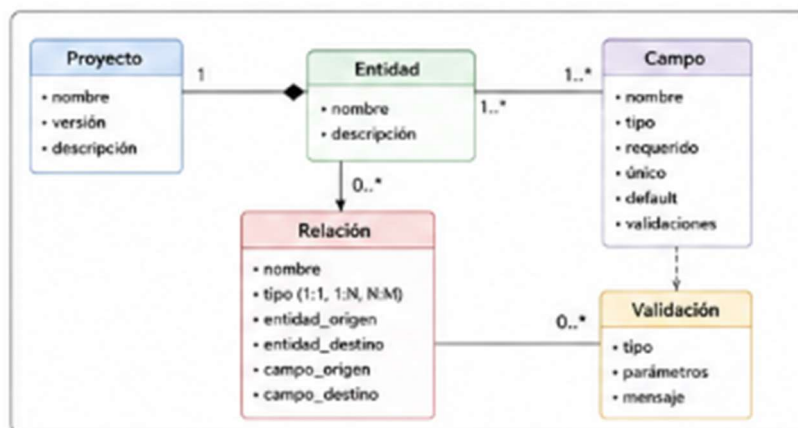


Ilustración 3. Metamodelo de la especificación del dominio.

4.2.1 Formato JSON

El sistema desarrollado utiliza especificaciones estructuradas en formato JSON como mecanismo principal para describir las características de las aplicaciones que posteriormente serán generadas de forma automática.

La elección de JSON (JavaScript Object Notation) se fundamenta en diversos factores. En primer lugar, se trata de un formato ampliamente utilizado en el intercambio de información entre aplicaciones y servicios. Además, presenta una sintaxis sencilla, legible tanto para humanos como para máquinas y dispone de soporte nativo en la mayoría de lenguajes y herramientas modernas.

La especificación JSON diseñada para este trabajo actúa como una representación abstracta del dominio de aplicación. A través de ella es posible describir entidades, atributos, restricciones de validación y relaciones sin necesidad de implementar directamente el código fuente correspondiente.

La estructura general de la especificación se compone de dos elementos principales:

- Información general del proyecto.
- Definición de entidades.

Cada entidad incluye la descripción de sus atributos y propiedades asociadas, permitiendo al sistema generar automáticamente los diferentes componentes necesarios para construir la aplicación backend.

A partir de esta representación, el generador es capaz de producir automáticamente los modelos de datos, validaciones, operaciones CRUD y demás componentes que conforman la aplicación resultante.

Este enfoque permite separar claramente la descripción del dominio de la implementación técnica, favoreciendo la reutilización y la automatización del proceso de desarrollo.

4.2.2 Definición de entidades

Las entidades constituyen el elemento fundamental de la especificación utilizada por el sistema. Cada entidad representa un concepto del dominio de aplicación y actúa como base para la generación automática de los diferentes componentes software asociados.

Desde el punto de vista conceptual, una entidad puede entenderse como una representación estructurada de un objeto o elemento relevante dentro del sistema. Ejemplos de entidades utilizadas durante el desarrollo del caso de estudio son *Book*, *Author* y *Publisher*, correspondientes a los distintos elementos presentes en una aplicación de gestión bibliográfica.

Cada entidad se define mediante un nombre único y una colección de atributos que describen sus características. Estos atributos contienen la información necesaria para generar posteriormente modelos de datos, validaciones, mecanismos de persistencia y operaciones CRUD.

La definición de una entidad dentro de la especificación JSON sigue una estructura sencilla basada en dos componentes principales:

- Nombre de la entidad.
- Conjunto de campos asociados.

Cada campo incluido en la entidad describe una característica concreta del objeto representado. Entre las propiedades soportadas por el sistema se encuentran:

- Nombre del atributo;
- Tipo de dato;
- Obligatoriedad;
- Restricciones de validación;
- Definición de clave primaria;
- Relaciones con otras entidades.

A partir de esta descripción, el generador construye automáticamente los modelos Pydantic utilizados para la validación de datos y los modelos SQLAlchemy encargados de gestionar la persistencia en base de datos.

Durante el desarrollo del caso de estudio se definieron tres entidades principales: *Book*, *Author* y *Publisher*. Estas entidades permitieron validar la capacidad del sistema para gestionar estructuras de datos independientes y relaciones entre diferentes elementos del dominio.

La utilización de entidades como unidad básica de modelado permite mantener una representación clara y coherente del dominio de aplicación, facilitando tanto la comprensión de la especificación como la generación automática de los distintos componentes software asociados.

4.2.3 Definición de validaciones

Además de la definición de entidades y atributos, la especificación diseñada permite incorporar restricciones de validación directamente sobre los campos del modelo. Estas restricciones proporcionan información adicional que posteriormente es utilizada por el generador para producir mecanismos automáticos de validación dentro de la aplicación resultante.

La incorporación de validaciones a nivel de especificación persigue dos objetivos principales. En primer lugar, garantizar la integridad de los datos gestionados por la API. En segundo lugar, evitar la necesidad de implementar manualmente reglas de validación repetitivas para cada entidad generada.

Las restricciones se describen mediante propiedades asociadas a cada atributo dentro de la especificación JSON. Dependiendo del tipo de dato, pueden definirse diferentes mecanismos de validación.

Para atributos de tipo cadena (*string*), el sistema permite especificar restricciones relacionadas con la obligatoriedad del campo y con la longitud mínima y máxima permitida.

Para atributos numéricos es posible definir límites mínimos y máximos que restringen los valores aceptados por la aplicación.

Estas restricciones son interpretadas automáticamente durante el proceso de generación y transformadas en validaciones implementadas mediante Pydantic.

Por ejemplo, la definición anterior produce automáticamente un modelo capaz de verificar que los datos recibidos cumplen las condiciones establecidas antes de que sean procesados por la aplicación.

Este enfoque ofrece diversas ventajas. En primer lugar, centraliza las reglas de validación dentro de la especificación del dominio. En segundo lugar, garantiza que todas las entidades generadas siguen criterios homogéneos de validación. Finalmente, reduce significativamente la cantidad de código manual necesario para implementar mecanismos de control de datos.

La capacidad de definir validaciones directamente en la especificación constituye uno de los elementos clave del sistema desarrollado, ya que permite generar aplicaciones backend más robustas y coherentes sin incrementar la complejidad de la descripción proporcionada por el usuario.

4.2.4 Definición de relaciones

Además de la definición de entidades independientes, el sistema permite describir relaciones entre distintos elementos del dominio mediante mecanismos de referencia incluidos en la propia especificación JSON.

La incorporación de relaciones resulta fundamental para representar adecuadamente escenarios reales, donde los datos suelen encontrarse conectados entre sí mediante asociaciones lógicas y dependencias estructurales.

Con el objetivo de soportar este tipo de situaciones, la especificación fue ampliada para permitir la definición de claves foráneas (*Foreign Keys*) que establecen vínculos entre entidades.

La representación de una relación se realiza mediante atributos especiales que indican la entidad de destino asociada a un determinado campo.

En este ejemplo, el atributo *author_id* actúa como referencia hacia una instancia de la entidad *Author*. De forma similar, pueden establecerse relaciones con otras entidades del sistema.

Durante el proceso de generación, esta información es interpretada por el generador y utilizada para producir automáticamente los mecanismos necesarios para gestionar la relación dentro de la aplicación resultante.

Entre los elementos generados se incluyen:

- claves foráneas en los modelos de persistencia;
- asociaciones ORM mediante SQLAlchemy;
- navegación relacional entre objetos;
- estructuras de respuesta enriquecidas con información relacionada.

Por ejemplo, la especificación anterior permite generar automáticamente una relación ORM similar a la siguiente:

Gracias a esta transformación automática, los objetos generados pueden acceder directamente a la información relacionada sin necesidad de implementar manualmente consultas adicionales.

Asimismo, el sistema permite generar respuestas enriquecidas que incluyen datos pertenecientes a entidades asociadas. De este modo, al recuperar un libro desde la API, es posible obtener simultáneamente la información de su autor y editorial.

La incorporación de relaciones dentro de la especificación permite incrementar significativamente la capacidad expresiva del modelo y aproximar las aplicaciones generadas a escenarios de desarrollo reales.

Además, esta funcionalidad demuestra que el sistema es capaz de generar no solo estructuras de datos independientes, sino también modelos de dominio con interacciones complejas entre entidades, ampliando considerablemente el alcance de la solución propuesta.

4.3 Parser de especificaciones

Una vez definida la estructura de la especificación de entrada, el siguiente componente fundamental del sistema es el módulo encargado de interpretar dicha información y prepararla para el proceso de generación.

Este módulo, denominado *parser de especificaciones*, actúa como puente entre la descripción abstracta proporcionada por el usuario y el motor de generación automática. Su función principal consiste en leer la especificación JSON, verificar su estructura y transformar la información contenida en una representación interna utilizable por el resto de componentes del sistema.

La separación entre el proceso de lectura y el proceso de generación proporciona diversas ventajas arquitectónicas. En primer lugar, permite desacoplar la interpretación de la especificación de los mecanismos de generación de código. En segundo lugar, facilita la evolución futura del sistema, ya que pueden incorporarse nuevos formatos de entrada o nuevas capacidades de generación sin modificar significativamente el resto de componentes.

Dentro de la arquitectura propuesta, el parser constituye el primer elemento ejecutado tras la recepción de una especificación. Su salida es utilizada posteriormente por el motor de generación basado en plantillas Jinja2 para producir los diferentes artefactos software que conforman la aplicación backend resultante.

Los apartados siguientes describen con detalle el proceso de lectura de la especificación y la construcción de la representación interna utilizada durante la generación automática.

4.3.1 Lectura del JSON

El proceso de generación comienza con la lectura de la especificación JSON proporcionada por el usuario. Este documento contiene toda la información necesaria para describir la aplicación que posteriormente será generada.

La implementación desarrollada utiliza las capacidades nativas de Python para cargar y procesar documentos JSON. Durante esta fase, el sistema accede al fichero especificado, interpreta su contenido y transforma la representación textual en estructuras de datos internas basadas en diccionarios y listas.

Una vez completada la lectura, el sistema dispone de acceso directo a todos los elementos definidos dentro de la especificación, incluyendo información del proyecto, entidades, atributos, validaciones y relaciones.

La utilización de JSON como formato de entrada simplifica considerablemente este proceso debido a la compatibilidad nativa existente entre dicho formato y las estructuras de datos proporcionadas por Python.

Además, esta aproximación permite mantener una separación clara entre los datos de configuración y la lógica de generación, favoreciendo la reutilización del sistema para diferentes dominios de aplicación.

El resultado de esta etapa es una representación estructurada de la especificación que servirá como entrada para las fases posteriores del proceso de generación.

4.3.2 Construcción del modelo interno

Una vez cargada la especificación JSON, el sistema realiza una fase adicional de procesamiento orientada a preparar la información para su utilización por el motor de generación.

Aunque la estructura obtenida tras la lectura del documento ya contiene toda la información necesaria para describir la aplicación, resulta conveniente enriquecer determinados elementos antes de utilizarlos dentro de las plantillas de generación. Este proceso da lugar a una representación interna más adecuada para la producción automática de código.

La principal transformación realizada consiste en la adaptación de los tipos definidos en la especificación a los tipos equivalentes utilizados por Python. Para ello, el sistema incorpora una tabla de correspondencias que permite traducir los tipos declarados por el usuario a representaciones compatibles con los modelos generados.

Gracias a esta transformación, las plantillas de generación pueden acceder directamente a los tipos adecuados sin necesidad de incorporar lógica compleja dentro del propio código de plantilla.

Además de la conversión de tipos, esta representación interna constituye un punto central donde podrían incorporarse futuras transformaciones, validaciones adicionales o mecanismos de enriquecimiento más avanzados.

Una vez completado este proceso, cada entidad contiene toda la información necesaria para generar automáticamente los diferentes componentes de la aplicación. Esta representación enriquecida es posteriormente suministrada al motor de generación basado en Jinja2, que utilizará los datos procesados para producir los artefactos software correspondientes.

La existencia de este modelo interno proporciona una clara separación entre la especificación de entrada y el proceso de generación, mejorando la mantenibilidad del sistema y facilitando futuras ampliaciones de la arquitectura.

4.4 Motor de generación

Una vez procesada la especificación de entrada y construida la representación interna del modelo, el siguiente paso consiste en generar automáticamente los distintos componentes software que conformarán la aplicación resultante.

Para llevar a cabo esta tarea se desarrolló un motor de generación basado en plantillas, cuyo objetivo es transformar la información abstracta obtenida del dominio en código fuente ejecutable. Este enfoque permite automatizar la producción de múltiples artefactos software manteniendo una clara separación entre la lógica de generación y el contenido generado.

La arquitectura adoptada se fundamenta en la utilización de plantillas parametrizadas que actúan como modelos de código reutilizables. Estas plantillas reciben información procedente de la especificación procesada y producen automáticamente los diferentes archivos que constituyen la aplicación backend.

Entre los componentes generados se encuentran:

- modelos Pydantic;
- routers FastAPI;
- modelos SQLAlchemy;
- configuración de persistencia;
- pruebas automatizadas;
- artefactos Docker.

La utilización de un motor de generación basado en plantillas aporta diversas ventajas. En primer lugar, facilita la reutilización de estructuras comunes entre diferentes entidades. En segundo lugar, simplifica el mantenimiento del sistema al centralizar la lógica de generación. Finalmente, permite incorporar nuevas capacidades mediante la creación de plantillas adicionales sin necesidad de modificar significativamente la arquitectura existente.

Los siguientes apartados describen con detalle la tecnología seleccionada para la generación, el flujo seguido durante el proceso y la organización interna de las plantillas utilizadas.

4.4.1 Uso de Jinja2

La generación automática de código se implementó utilizando Jinja2, un motor de plantillas ampliamente utilizado dentro del ecosistema Python.

Jinja2 permite definir documentos parametrizados que contienen fragmentos de código estático combinados con expresiones dinámicas. Durante el proceso de generación, estas expresiones son sustituidas por valores procedentes de la especificación procesada, produciendo como resultado archivos completamente funcionales.

La principal ventaja de este enfoque radica en la separación entre la estructura del código generado y los datos utilizados para producirlo. De este modo, la lógica de generación permanece centralizada dentro de las plantillas mientras que las características específicas de cada aplicación se obtienen directamente de la especificación de entrada.

Para inicializar el entorno de generación, el sistema crea una instancia de Jinja2 asociada al directorio que contiene las plantillas.

Una vez creado el entorno, cada plantilla puede cargarse individualmente y utilizarse para generar uno o varios archivos.

Por ejemplo, la generación de modelos Pydantic se realiza cargando la plantilla correspondiente y suministrando como contexto la información asociada a cada entidad.

El resultado obtenido es un documento de texto que contiene el código fuente final generado automáticamente.

La utilización de Jinja2 permitió implementar un sistema flexible, fácilmente extensible y capaz de producir diferentes tipos de artefactos software a partir de una única representación del dominio. Esta característica resulta especialmente importante en un contexto de generación automática, donde múltiples componentes deben mantenerse sincronizados respecto a una misma especificación de entrada.

4.4.2 Flujo de generación

Una vez configurado el entorno de plantillas y construida la representación interna de la especificación, el sistema inicia el proceso de generación automática de código.

El flujo de generación desarrollado sigue una secuencia estructurada de etapas, donde cada componente se genera de forma independiente a partir de la misma especificación de entrada. Esta aproximación garantiza la consistencia entre todos los artefactos producidos y facilita la incorporación de nuevas capacidades al sistema.

El proceso comienza con la carga de la especificación JSON y la construcción del modelo interno enriquecido. Posteriormente, el motor de generación recorre las entidades definidas en la especificación y aplica las plantillas correspondientes para producir los diferentes componentes software.

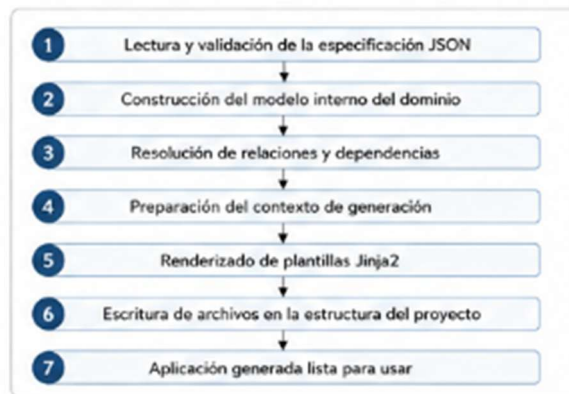


Ilustración 4. Proceso de generación de la aplicación.

Cada uno de estos componentes es generado mediante funciones especializadas dentro del módulo principal del generador. Estas funciones se encargan de seleccionar la plantilla adecuada, suministrar el contexto necesario y escribir el resultado en la estructura de directorios correspondiente.

Esta estrategia permite que todos los elementos de la aplicación sean producidos automáticamente a partir de una única fuente de información.

Además, el diseño modular del flujo facilita la evolución futura del sistema. La incorporación de nuevos artefactos generados únicamente requiere implementar nuevas funciones de generación y asociarlas a las plantillas correspondientes, sin necesidad de modificar significativamente el resto de la arquitectura.

Como resultado, el sistema es capaz de transformar una especificación compacta y declarativa en una aplicación backend completa compuesta por múltiples componentes coordinados entre sí.

4.4.3 Organización de plantillas

El sistema de generación desarrollado se basa en una colección de plantillas Jinja2 especializadas, cada una de ellas responsable de producir un tipo concreto de artefacto software.

Esta organización modular permite separar claramente la lógica de generación asociada a cada componente de la aplicación, facilitando el mantenimiento y la evolución del sistema. Asimismo, posibilita incorporar nuevas capacidades mediante la creación de plantillas adicionales sin necesidad de modificar el núcleo del generador.

Todas las plantillas se almacenan dentro de un directorio específico denominado *templates*, que constituye el repositorio central de conocimiento utilizado durante el proceso de generación.

Cada plantilla tiene una responsabilidad concreta dentro del proceso de generación:

model.j2

Genera los modelos Pydantic utilizados para representar y validar los datos intercambiados por la API.

A partir de las entidades definidas en la especificación, esta plantilla produce automáticamente clases Python con los atributos y validaciones correspondientes.

router.j2

Genera los routers FastAPI encargados de exponer los endpoints REST de cada entidad.

La plantilla incorpora automáticamente operaciones CRUD, integración con persistencia y mecanismos de acceso a datos.

main.j2

Genera el archivo principal de la aplicación.

Este componente se encarga de inicializar FastAPI, registrar los routers generados y configurar el punto de entrada de la aplicación.

database.j2

Genera la configuración necesaria para gestionar la conexión con la base de datos y proporcionar sesiones SQLAlchemy a los distintos componentes del sistema.

sqlalchemy_model.j2

Genera los modelos ORM utilizados para la persistencia de datos.

A partir de la información contenida en la especificación, esta plantilla produce clases SQLAlchemy capaces de representar entidades, claves primarias y relaciones entre tablas.

test_router.j2

Genera automáticamente pruebas básicas utilizando Pytest.

Estas pruebas permiten verificar el correcto funcionamiento de los endpoints generados y constituyen un mecanismo inicial de validación de la aplicación resultante.

Dockerfile.j2

Genera automáticamente el archivo Dockerfile necesario para construir imágenes contenedorizadas de la aplicación.

Esta funcionalidad facilita el despliegue y la distribución de las APIs generadas.

requirements.j2

Genera el fichero de dependencias Python utilizado para instalar automáticamente las bibliotecas necesarias para ejecutar la aplicación generada.

Ventajas de la organización modular

La utilización de plantillas independientes proporciona varias ventajas arquitectónicas.

En primer lugar, facilita el mantenimiento del sistema, ya que cada componente puede evolucionar de forma aislada. En segundo lugar, simplifica la incorporación de nuevas capacidades mediante la creación de plantillas adicionales.

Finalmente, favorece la reutilización de estructuras comunes y reduce la duplicación de lógica de generación.

Gracias a esta organización, el generador puede producir aplicaciones backend completas a partir de una única especificación, manteniendo una arquitectura flexible y fácilmente extensible.

4.5 Arquitectura de código generado

El objetivo principal del sistema desarrollado consiste en transformar una especificación estructurada en una aplicación backend completamente funcional. Para lograrlo, el motor de generación produce automáticamente distintos tipos de componentes software que colaboran entre sí para formar la aplicación final.

La arquitectura del código generado se diseñó siguiendo una estructura modular inspirada en las prácticas habituales del desarrollo backend moderno. Cada componente generado posee una responsabilidad claramente definida, favoreciendo la separación de responsabilidades, la mantenibilidad y la escalabilidad de la solución resultante.

Los artefactos producidos por el sistema se organizan en diferentes módulos especializados, incluyendo modelos de validación, routers REST, componentes de persistencia, modelos ORM y archivos de configuración.

Esta organización permite mantener una separación clara entre los distintos niveles de la aplicación y facilita la evolución futura del software generado.

En los apartados siguientes se describe con detalle cada uno de los componentes producidos automáticamente por el sistema.

4.5.1 Modelos Pydantic

Los modelos Pydantic constituyen uno de los primeros componentes generados automáticamente por el sistema. Su función principal consiste en representar las estructuras de datos intercambiadas por la API y garantizar que la información recibida cumple las restricciones definidas en la especificación.

Para cada entidad descrita en el documento JSON, el generador produce automáticamente una clase Python basada en Pydantic. Estas clases contienen los atributos definidos por el usuario y las validaciones asociadas a cada uno de ellos.

Cuando la especificación incorpora restricciones de validación, estas son transformadas automáticamente en configuraciones compatibles con Pydantic.

De este modo, atributos definidos mediante propiedades como *required*, *min_length*, *max_length*, *minimum* o *maximum* generan mecanismos automáticos de control que verifican la validez de los datos recibidos por la API.

La utilización de modelos Pydantic aporta diversas ventajas:

- validación automática de entradas;
- generación de esquemas OpenAPI;
- documentación automática;
- detección temprana de errores;
- integración directa con FastAPI.

Gracias a este enfoque, las aplicaciones generadas incorporan mecanismos robustos de validación sin necesidad de implementar código adicional por parte del desarrollador.

Además, la generación automática garantiza que todos los modelos mantienen una estructura coherente y alineada con la especificación original, reduciendo la probabilidad de inconsistencias entre los distintos componentes del sistema.

4.5.2 Routers FastAPI

Los routers FastAPI constituyen el mecanismo principal mediante el cual las aplicaciones generadas exponen su funcionalidad a clientes externos. Su responsabilidad consiste en definir los endpoints REST asociados a cada entidad y gestionar las operaciones realizadas sobre los datos.

Para cada entidad descrita en la especificación, el sistema genera automáticamente un router independiente que incorpora las operaciones CRUD básicas necesarias para interactuar con los recursos correspondientes.

La estructura generada sigue las convenciones habituales de FastAPI, utilizando el componente *APIRouter* para organizar los endpoints de forma modular.

A partir de la especificación original, el generador produce automáticamente los siguientes tipos de operaciones:

- Obtención de todos los elementos (*GET*).
- Creación de nuevos registros (*POST*).
- Actualización de elementos existentes (*PUT*).
- Eliminación de registros (*DELETE*).

Estas operaciones se generan para todas las entidades definidas en la especificación, evitando la necesidad de implementar manualmente cada endpoint.

La integración con FastAPI proporciona diversas ventajas adicionales. Entre ellas destacan la validación automática de los datos recibidos, la serialización de respuestas, la generación automática de documentación OpenAPI y la disponibilidad inmediata de una interfaz Swagger para realizar pruebas.

Durante las primeras versiones del sistema, los routers utilizaban estructuras temporales en memoria para almacenar los datos generados. Posteriormente, la arquitectura evolucionó hacia una solución basada en persistencia mediante SQLAlchemy y SQLite, permitiendo que los endpoints interactuaran directamente con la base de datos.

Gracias a esta evolución, los routers generados pasaron de actuar como simples demostradores funcionales a convertirse en componentes capaces de gestionar información persistente y relaciones entre entidades.

La generación automática de routers constituye uno de los elementos más relevantes del sistema desarrollado, ya que permite obtener una API REST completamente funcional a partir de una descripción estructurada del dominio sin necesidad de implementar manualmente los endpoints asociados.

4.5.3 Configuración de persistencia

Una de las funcionalidades incorporadas durante la evolución del sistema fue la generación automática de los componentes necesarios para gestionar la persistencia de datos.

En las primeras versiones del prototipo, los endpoints generados utilizaban estructuras temporales en memoria para almacenar la información gestionada por la API. Aunque este enfoque resultaba suficiente para validar el funcionamiento básico de los routers y operaciones CRUD, presentaba limitaciones importantes, ya que los datos se perdían al finalizar la ejecución de la aplicación.

Con el objetivo de obtener aplicaciones backend funcionales y más próximas a escenarios reales, se incorporó soporte para persistencia mediante SQLite y SQLAlchemy.

Para ello, el generador produce automáticamente un archivo denominado *database.py*, encargado de centralizar la configuración de acceso a la base de datos y proporcionar los mecanismos necesarios para gestionar sesiones de trabajo.

La configuración generada incluye la creación del motor de conexión (*engine*), la definición de la sesión SQLAlchemy y la declaración de la clase base utilizada por los modelos ORM.

Además, el sistema genera automáticamente una función auxiliar encargada de proporcionar sesiones de base de datos a los distintos componentes de la aplicación.

Esta función es utilizada posteriormente por los routers generados mediante el mecanismo de inyección de dependencias proporcionado por FastAPI.

Gracias a esta integración, cada petición dispone de una sesión independiente de acceso a datos, garantizando una gestión adecuada de recursos y facilitando la interacción con la base de datos.

La generación automática de la configuración de persistencia ofrece varias ventajas. En primer lugar, elimina la necesidad de realizar configuraciones manuales repetitivas para cada proyecto generado. En segundo lugar, garantiza que todos los componentes utilizan una configuración homogénea. Finalmente, proporciona una base sólida para la implementación de operaciones CRUD persistentes y relaciones entre entidades.

La incorporación de esta capa de persistencia supuso un avance significativo dentro de la evolución del prototipo, permitiendo que las aplicaciones generadas almacenasen información de forma permanente y se aproximasen a las características habituales de los sistemas backend reales.

4.5.4 Modelos SQLAlchemy

Además de los modelos Pydantic utilizados para validación y transferencia de datos, el sistema genera automáticamente modelos ORM basados en SQLAlchemy para gestionar la persistencia de información en la base de datos.

Estos modelos constituyen la representación interna de las entidades dentro del sistema de almacenamiento y actúan como intermediarios entre los objetos Python y las tablas físicas existentes en la base de datos.

Para cada entidad definida en la especificación JSON, el generador produce automáticamente una clase SQLAlchemy independiente. Estas clases incluyen la definición de atributos, claves primarias, relaciones y restricciones necesarias para representar correctamente la estructura del dominio.

A diferencia de los modelos Pydantic, cuyo objetivo principal es validar los datos intercambiados por la API, los modelos SQLAlchemy se encargan de gestionar la persistencia y recuperación de información desde la base de datos.

Durante la evolución del proyecto se amplió el sistema para soportar también relaciones entre entidades. Como consecuencia, los modelos generados incorporan automáticamente claves foráneas y asociaciones ORM derivadas de la información contenida en la especificación.

De forma análoga, las relaciones con la entidad *Publisher* generan los mecanismos correspondientes para establecer asociaciones entre tablas.

Gracias a estas transformaciones, los objetos persistidos pueden acceder directamente a la información relacionada sin necesidad de implementar consultas manuales complejas.

La generación automática de modelos SQLAlchemy ofrece múltiples ventajas:

- Eliminación de código repetitivo;
- Consistencia entre el modelo lógico y la base de datos;
- Soporte automático para persistencia;
- Generación de relaciones ORM;
- Integración directa con FastAPI.

Además, dado que todos los modelos se producen a partir de una única especificación de entrada, se garantiza que la estructura de persistencia permanece sincronizada con el resto de componentes generados por el sistema.

La incorporación de esta capa ORM permitió evolucionar desde una solución basada exclusivamente en validación y exposición de endpoints hacia una arquitectura backend completa, capaz de gestionar datos persistentes y relaciones complejas entre entidades.

Como resultado, las aplicaciones generadas pueden operar sobre bases de datos reales manteniendo una correspondencia directa entre la especificación original y la estructura de almacenamiento producida automáticamente.

4.6 Soporte para persistencia

Uno de los objetivos fundamentales perseguidos durante el desarrollo del sistema consistió en generar aplicaciones backend capaces de almacenar información de forma permanente.

Las primeras versiones del prototipo utilizaban estructuras de datos temporales mantenidas en memoria para implementar las operaciones CRUD generadas automáticamente. Aunque este enfoque permitía validar el funcionamiento básico de los endpoints, presentaba una limitación importante: toda la información se perdía al finalizar la ejecución de la aplicación.

Con el fin de aproximar las aplicaciones generadas a escenarios de desarrollo reales, se incorporó una capa completa de persistencia basada en SQLite y SQLAlchemy. Esta ampliación permitió almacenar datos de forma permanente, gestionar relaciones entre entidades y proporcionar soporte para operaciones CRUD persistentes.

La integración de persistencia se diseñó de manera totalmente transparente para el usuario. A partir de una misma especificación JSON, el sistema es capaz de generar automáticamente los componentes necesarios para crear la base de datos, definir los modelos ORM y realizar operaciones de acceso a datos.

Los apartados siguientes describen las tecnologías empleadas y los mecanismos implementados para proporcionar persistencia automática dentro de las aplicaciones generadas.

4.6.1 SQLite

Como sistema de almacenamiento persistente se seleccionó SQLite, una base de datos relacional ligera ampliamente utilizada en aplicaciones de pequeño y mediano tamaño.

La elección de SQLite se fundamenta principalmente en su simplicidad de uso y despliegue. A diferencia de otros sistemas gestores de bases de datos, SQLite no requiere procesos servidores independientes ni configuraciones complejas, ya que toda la información se almacena directamente en un único fichero.

Esta característica resulta especialmente adecuada para los objetivos del proyecto, ya que permite generar aplicaciones completamente funcionales sin imponer requisitos adicionales al usuario.

La configuración de conexión se genera automáticamente dentro del archivo de persistencia producido por el sistema.

Gracias a esta configuración, las aplicaciones generadas crean automáticamente el fichero de base de datos cuando se ejecutan por primera vez, simplificando considerablemente el proceso de despliegue y validación.

Aunque SQLite presenta limitaciones en escenarios de alta concurrencia o grandes volúmenes de datos, sus características resultan más que suficientes para validar la viabilidad de la solución propuesta y demostrar las capacidades del sistema de generación automática.

Además, la utilización de SQLAlchemy como capa de abstracción permite que futuras versiones del generador puedan adaptarse fácilmente a otros motores de bases de datos sin modificar significativamente la arquitectura general del sistema.

4.6.2 SQLAlchemy ORM

Aunque SQLite proporciona el mecanismo físico de almacenamiento de información, el acceso a los datos dentro de las aplicaciones generadas se realiza mediante SQLAlchemy, una de las bibliotecas ORM (Object Relational Mapping) más utilizadas dentro del ecosistema Python.

El uso de un ORM permite trabajar con objetos Python en lugar de interactuar directamente con consultas SQL. De este modo, las entidades definidas en la especificación se representan mediante clases, mientras que los registros almacenados en la base de datos se gestionan como instancias de dichas clases.

Esta aproximación aporta diversas ventajas desde el punto de vista del desarrollo software. En primer lugar, reduce la cantidad de código SQL necesario para implementar operaciones de acceso a datos. En segundo lugar, mejora la legibilidad y mantenibilidad de la aplicación. Finalmente, facilita la gestión de relaciones entre entidades y la portabilidad entre diferentes motores de bases de datos.

Dentro de la solución propuesta, SQLAlchemy desempeña un papel central en la arquitectura generada. Los modelos ORM producidos automáticamente actúan como intermediarios entre la aplicación FastAPI y la base de datos SQLite.

La integración se realiza mediante sesiones SQLAlchemy generadas automáticamente por el sistema. Estas sesiones proporcionan acceso a las operaciones necesarias para consultar, insertar, modificar y eliminar registros.

Por ejemplo, la recuperación de todos los elementos de una entidad se implementa mediante operaciones ORM similares a la siguiente:

Del mismo modo, la creación de nuevos registros se realiza utilizando objetos ORM generados automáticamente:

Además de simplificar el acceso a datos, SQLAlchemy proporciona mecanismos avanzados para gestionar asociaciones entre entidades. Gracias a ello, las relaciones definidas en la especificación JSON pueden transformarse automáticamente en asociaciones ORM capaces de navegar entre objetos relacionados.

Esta capacidad fue especialmente relevante durante la evolución del proyecto, ya que permitió incorporar soporte para claves foráneas, relaciones entre entidades y respuestas enriquecidas sin necesidad de implementar consultas manuales complejas.

La utilización de SQLAlchemy contribuye significativamente a la flexibilidad de la arquitectura generada. Aunque el prototipo utiliza SQLite como sistema de almacenamiento principal, la capa ORM facilita la adaptación futura del sistema a motores de bases de datos más avanzados, como PostgreSQL o MySQL, minimizando el impacto sobre el código generado.

Por tanto, SQLAlchemy no solo actúa como mecanismo de persistencia, sino también como elemento de abstracción que simplifica el desarrollo y amplía las posibilidades de evolución futura de la solución propuesta.

4.6.3 CRUD persistente

La incorporación de persistencia mediante SQLite y SQLAlchemy permitió evolucionar significativamente las capacidades de las aplicaciones generadas. Como consecuencia, las operaciones CRUD dejaron de trabajar sobre estructuras temporales en memoria para interactuar directamente con una base de datos relacional.

En las primeras versiones del sistema, las operaciones generadas utilizaban listas Python para almacenar la información procesada por la API.

Aunque esta implementación permitía validar el funcionamiento básico de los endpoints generados, presentaba limitaciones importantes. Los datos no se conservaban entre ejecuciones y no era posible gestionar relaciones complejas ni realizar consultas avanzadas.

Para superar estas limitaciones, se modificó la arquitectura generada incorporando operaciones basadas en SQLAlchemy. Como resultado, los endpoints pasaron a interactuar directamente con los modelos ORM y la base de datos persistente.

Del mismo modo, la creación de nuevos registros se implementa utilizando objetos ORM generados automáticamente.

La utilización de operaciones persistentes proporciona múltiples ventajas frente a la implementación inicial basada en memoria:

- Conservación permanente de la información;
- Soporte para relaciones entre entidades;
- Acceso eficiente a grandes volúmenes de datos;
- Integración con mecanismos ORM;
- Comportamiento más próximo a aplicaciones reales.

Además, este enfoque permitió completar la implementación de las operaciones CRUD fundamentales:

- **Create:** inserción de nuevos registros.
- **Read:** recuperación de información almacenada.
- **Update:** modificación de registros existentes.
- **Delete:** eliminación de registros persistidos.

La validación experimental realizada durante el desarrollo confirmó el correcto funcionamiento de estas operaciones mediante pruebas realizadas a través de Swagger y Pytest.

La incorporación de CRUD persistente representa uno de los hitos más relevantes dentro de la evolución del sistema, ya que transformó el prototipo inicial en una solución capaz de generar aplicaciones backend completamente funcionales y con persistencia real de datos.

4.7 Gestión de relaciones

Una de las funcionalidades más avanzadas incorporadas al sistema desarrollado consiste en la generación automática de relaciones entre entidades.

En aplicaciones reales, los datos rara vez se encuentran completamente aislados. Habitualmente existen asociaciones que conectan diferentes elementos del dominio, permitiendo representar dependencias, jerarquías y vínculos lógicos entre objetos.

Con el objetivo de aproximar las aplicaciones generadas a escenarios de desarrollo reales, el sistema fue ampliado para soportar relaciones entre entidades definidas directamente dentro de la especificación JSON.

A partir de esta información, el generador es capaz de producir automáticamente los mecanismos necesarios para representar dichas relaciones tanto a nivel de base de datos como a nivel de objetos dentro de la aplicación.

La solución implementada incorpora soporte para:

- Claves foráneas (*Foreign Keys*);
- Relaciones ORM mediante SQLAlchemy;
- Navegación entre entidades relacionadas;
- Respuestas enriquecidas con información asociada.

Gracias a estas capacidades, las aplicaciones generadas pueden gestionar estructuras de datos significativamente más complejas que las soportadas por las primeras versiones del prototipo.

Los apartados siguientes describen los mecanismos utilizados para implementar estas funcionalidades.

4.7.1 Foreign Keys

El primer paso para incorporar relaciones entre entidades consistió en la generación automática de claves foráneas dentro de los modelos de persistencia.

Las claves foráneas constituyen un mecanismo fundamental en las bases de datos relacionales, ya que permiten establecer vínculos entre registros pertenecientes a tablas diferentes.

Para soportar esta funcionalidad, la especificación JSON fue ampliada con atributos capaces de indicar referencias hacia otras entidades.

Durante el proceso de generación, esta información es interpretada automáticamente y transformada en una restricción de integridad referencial dentro del modelo SQLAlchemy correspondiente.

De forma análoga, el sistema genera automáticamente las referencias necesarias para cualquier otra relación definida en la especificación.

La utilización de claves foráneas aporta diversas ventajas:

- Mantenimiento de la integridad de los datos;
- Prevención de referencias inválidas;
- Representación explícita de relaciones;
- Compatibilidad con mecanismos ORM.

Gracias a esta funcionalidad, el sistema es capaz de trasladar automáticamente las relaciones definidas a nivel conceptual hacia la estructura física de la base de datos generada.

4.7.2 Relaciones ORM

Aunque las claves foráneas permiten establecer vínculos entre tablas a nivel de base de datos, trabajar exclusivamente con identificadores resulta poco práctico desde el punto de vista del desarrollo de aplicaciones.

Con el objetivo de simplificar el acceso a la información relacionada, el sistema genera automáticamente asociaciones ORM utilizando los mecanismos proporcionados por SQLAlchemy.

Estas asociaciones permiten navegar directamente entre objetos relacionados sin necesidad de realizar consultas SQL explícitas ni gestionar manualmente las uniones entre tablas.

A partir de una relación definida en la especificación JSON, el generador produce automáticamente las estructuras ORM correspondientes dentro de los modelos persistentes.

En este ejemplo, el atributo *author_id* representa la clave foránea almacenada físicamente en la base de datos, mientras que el atributo *author* proporciona acceso directo al objeto relacionado.

De forma similar, el sistema genera automáticamente las relaciones necesarias para cualquier otra entidad referenciada en la especificación.

La principal ventaja de este enfoque radica en que los desarrolladores pueden trabajar con objetos completos en lugar de gestionar únicamente identificadores numéricos. Esto simplifica considerablemente la implementación de operaciones de acceso a datos y mejora la legibilidad del código generado.

Además, SQLAlchemy se encarga automáticamente de resolver las asociaciones necesarias cuando una entidad relacionada es consultada, ocultando gran parte de la complejidad inherente a las bases de datos relacionales.

La incorporación automática de relaciones ORM aporta diversas ventajas:

- Simplificación del acceso a datos relacionados;
- Reducción de consultas manuales;
- Mejora de la legibilidad del código;
- Integración transparente con SQLAlchemy;
- Soporte para modelos de dominio más complejos.

Esta funcionalidad representa una evolución significativa respecto a las primeras versiones del sistema, ya que permite generar aplicaciones capaces de gestionar relaciones entre entidades de forma natural y coherente.

Como resultado, las aplicaciones producidas por el generador no solo almacenan datos relacionados mediante claves foráneas, sino que también proporcionan mecanismos de navegación orientados a objetos equivalentes a los utilizados en desarrollos backend profesionales.

4.7.3 Navegación relacional

La incorporación de claves foráneas y relaciones ORM permitió implementar una funcionalidad adicional orientada a mejorar la calidad de las respuestas generadas por la API: la navegación relacional.

En las primeras versiones del sistema, las respuestas devueltas por los endpoints contenían únicamente los identificadores asociados a las entidades relacionadas. Por ejemplo, un libro almacenaba exclusivamente los valores *author_id* y *publisher_id*.

Aunque esta representación resulta válida desde el punto de vista funcional, obliga a los clientes de la API a realizar consultas adicionales para recuperar la información completa de las entidades relacionadas.

Con el objetivo de proporcionar respuestas más expresivas y próximas a las utilizadas en aplicaciones reales, el sistema fue ampliado para aprovechar las capacidades de navegación proporcionadas por SQLAlchemy.

Gracias a las relaciones ORM generadas automáticamente, los objetos persistentes pueden acceder directamente a la información asociada a otras entidades. Esto permite construir respuestas enriquecidas que incluyen datos relacionados dentro de la misma estructura devuelta por la API.

Este comportamiento fue validado durante las pruebas realizadas mediante Swagger UI, comprobando que las entidades relacionadas eran recuperadas correctamente y devueltas como parte de la respuesta final.

La navegación relacional aporta diversas ventajas:

- Reducción del número de consultas necesarias;
- Simplificación de los clientes consumidores de la API;
- Respuestas más completas y expresivas;
- Mejor representación del dominio de aplicación;
- Aprovechamiento de las capacidades ORM generadas automáticamente.

Desde la perspectiva del usuario final, esta funcionalidad permite trabajar con estructuras de datos más ricas y cercanas a los modelos de negocio reales.

Además, demuestra que el sistema desarrollado es capaz no solo de generar operaciones CRUD básicas, sino también de producir aplicaciones backend capaces de gestionar relaciones complejas entre entidades y exponerlas mediante interfaces REST modernas.

La incorporación de navegación relacional constituye una de las funcionalidades más avanzadas implementadas durante el proyecto y representa un paso significativo hacia la generación automática de aplicaciones backend completas y funcionales.

4.8 Calidad software

Además de la generación automática de componentes funcionales, durante el desarrollo del sistema se incorporaron mecanismos orientados a mejorar la calidad del software producido.

La calidad constituye un aspecto fundamental en cualquier proceso de desarrollo, especialmente cuando se utilizan técnicas de generación automática de código. La capacidad de producir aplicaciones funcionales resulta insuficiente si no se dispone de mecanismos que permitan verificar su correcto comportamiento y garantizar la coherencia de los artefactos generados.

Con este objetivo, el sistema fue ampliado para generar automáticamente pruebas básicas sobre los endpoints creados y para facilitar la validación funcional de las aplicaciones resultantes.

Los apartados siguientes describen las estrategias utilizadas para verificar el correcto funcionamiento del software generado.

4.8.1 Generación automática de pruebas

Con el objetivo de incorporar mecanismos básicos de aseguramiento de calidad, el sistema desarrollado genera automáticamente pruebas unitarias para cada una de las entidades definidas en la especificación.

Estas pruebas se producen mediante plantillas Jinja2 específicas y se implementan utilizando Pytest, uno de los frameworks de testing más utilizados dentro del ecosistema Python.

Para cada entidad generada, el sistema crea automáticamente un archivo de pruebas independiente dentro del directorio correspondiente.

Las pruebas generadas están orientadas a verificar el correcto funcionamiento de los endpoints principales de cada entidad.

Entre las validaciones realizadas se incluyen:

- Creación de nuevos registros;
- Recuperación de información;
- Comprobación de respuestas HTTP;
- Validación básica del comportamiento de la API.

La generación automática de pruebas proporciona diversas ventajas. En primer lugar, permite verificar rápidamente que la aplicación generada funciona correctamente tras su creación. En segundo lugar, facilita la detección temprana de errores durante la evolución del sistema. Finalmente, introduce prácticas básicas de calidad software dentro del proceso de generación automática.

Durante el desarrollo del proyecto, estas pruebas fueron utilizadas para validar las distintas funcionalidades incorporadas progresivamente, incluyendo persistencia, relaciones entre entidades y navegación relacional.

La integración de Pytest dentro del proceso de generación representa un paso adicional hacia la construcción de aplicaciones backend completas, incorporando no solo funcionalidad sino también mecanismos básicos de verificación automática.

4.8.2 Validación funcional

Además de las pruebas automatizadas generadas mediante Pytest, la validación del sistema se complementó mediante la ejecución práctica de las aplicaciones producidas y la comprobación manual de su comportamiento.

El objetivo de esta validación funcional consistió en verificar que los distintos componentes generados automáticamente interactuaban correctamente entre sí y que la aplicación resultante era capaz de proporcionar todas las funcionalidades previstas.

Para ello, las APIs generadas fueron desplegadas localmente utilizando Uvicorn como servidor ASGI y FastAPI como framework de ejecución.

Una vez iniciada la aplicación, se utilizó la documentación interactiva generada automáticamente por OpenAPI y accesible mediante Swagger UI para realizar pruebas sobre los distintos endpoints producidos por el sistema.

Durante esta fase se verificaron distintos aspectos del comportamiento de las aplicaciones generadas:

- Creación de registros mediante operaciones POST;
- Recuperación de información mediante operaciones GET;
- Actualización de datos mediante operaciones PUT;
- Eliminación de registros mediante operaciones DELETE;
- Persistencia correcta en SQLite;
- Funcionamiento de relaciones entre entidades;
- Generación de respuestas relacionales enriquecidas.

Especial atención se prestó a la validación de las relaciones entre entidades, comprobando que los modelos ORM generados eran capaces de recuperar correctamente la información asociada a autores y editoriales desde la entidad Book.

Asimismo, se verificó el correcto funcionamiento de los mecanismos automáticos de validación generados a partir de las restricciones definidas en la especificación JSON.

Como complemento a las pruebas manuales, se ejecutaron los conjuntos de pruebas generados automáticamente mediante Pytest. Los resultados obtenidos confirmaron el correcto funcionamiento de los endpoints implementados para las entidades utilizadas en el caso de estudio.

La combinación de pruebas automatizadas y validación funcional permitió comprobar que la arquitectura propuesta era capaz de generar aplicaciones backend operativas, coherentes y alineadas con los requisitos establecidos durante el desarrollo del proyecto.

Los resultados obtenidos durante esta fase constituyen la base para la evaluación experimental presentada posteriormente en el Capítulo 6, donde se analizará con mayor detalle el impacto de la generación automática sobre el proceso de desarrollo.

4.9 Soporte para despliegue

Además de generar automáticamente los componentes funcionales de la aplicación, el sistema incorpora mecanismos orientados a facilitar su ejecución y distribución.

En entornos de desarrollo reales, la generación del código fuente constituye únicamente una parte del proceso. También resulta necesario proporcionar configuraciones que permitan instalar dependencias, reproducir el entorno de ejecución y desplegar la aplicación de forma sencilla.

Con este objetivo, el generador fue ampliado para producir automáticamente los artefactos necesarios para la instalación y contenerización de las aplicaciones generadas.

Concretamente, el sistema genera un fichero de dependencias Python y un archivo Dockerfile capaces de automatizar la preparación del entorno de ejecución.

Los apartados siguientes describen ambos mecanismos.

4.9.1 Generación automática de dependencias

Las aplicaciones generadas dependen de diversas bibliotecas externas necesarias para proporcionar funcionalidades relacionadas con el desarrollo web, validación de datos, persistencia y pruebas.

Con el fin de simplificar la instalación del entorno de ejecución, el sistema genera automáticamente un fichero *requirements.txt* que contiene todas las dependencias necesarias para ejecutar la aplicación resultante.

La generación de este archivo se realiza mediante una plantilla Jinja2 específica, permitiendo mantener centralizada la definición de las bibliotecas requeridas.

La utilización de este fichero permite instalar automáticamente todas las dependencias mediante la herramienta *pip*.

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

La generación automática de dependencias aporta varias ventajas. En primer lugar, facilita la reproducibilidad de las aplicaciones generadas. En segundo lugar, simplifica la configuración inicial del entorno de desarrollo. Finalmente, reduce la probabilidad de errores derivados de instalaciones incompletas o inconsistentes.

Gracias a esta funcionalidad, cualquier aplicación producida por el sistema puede prepararse para su ejecución de forma rápida y homogénea.

4.9.2 Generación automática de *Dockerfile*

Con el objetivo de mejorar la portabilidad y facilitar el despliegue de las aplicaciones generadas, el sistema incorpora soporte para la generación automática de contenedores Docker.

Docker constituye una de las tecnologías más utilizadas actualmente para distribuir aplicaciones de forma reproducible e independiente del entorno de ejecución subyacente. Mediante el uso de contenedores, es posible garantizar que una aplicación se ejecutará de forma consistente en diferentes sistemas y plataformas.

Para soportar esta funcionalidad, el generador produce automáticamente un archivo *Dockerfile* a partir de una plantilla Jinja2 específica.

Este archivo contiene todas las instrucciones necesarias para construir una imagen ejecutable de la aplicación generada.

El proceso de despliegue puede realizarse mediante los comandos estándar proporcionados por Docker.

```
docker build -t generated-api .
```

```
docker run -p 8000:8000 generated-api
```

La incorporación de esta funcionalidad supone una mejora significativa respecto a versiones anteriores del sistema, ya que permite distribuir las aplicaciones generadas de forma sencilla y reproducible.

Además, la generación automática del *Dockerfile* constituye un ejemplo de cómo el sistema no se limita a producir código fuente, sino que también genera artefactos auxiliares necesarios para facilitar el ciclo completo de desarrollo y despliegue.

Como resultado, las aplicaciones producidas pueden ser ejecutadas tanto en entornos locales como en infraestructuras basadas en contenedores sin necesidad de configuraciones manuales adicionales.

4.10 Resumen de la arquitectura

En este capítulo se ha presentado el diseño de la solución propuesta para la generación automática de aplicaciones backend a partir de especificaciones estructuradas.

La arquitectura desarrollada se fundamenta en la utilización de documentos JSON como mecanismo de descripción del dominio de aplicación. A partir de estas especificaciones, el sistema es capaz de interpretar entidades, atributos, validaciones y relaciones, transformando dicha información en una representación interna utilizada durante el proceso de generación.

El núcleo de la solución está constituido por un motor de generación basado en plantillas Jinja2, responsable de producir automáticamente los diferentes componentes software que conforman la aplicación resultante. Entre los artefactos generados se incluyen modelos Pydantic, routers FastAPI, modelos SQLAlchemy, configuraciones de persistencia, pruebas automatizadas y mecanismos básicos de despliegue.

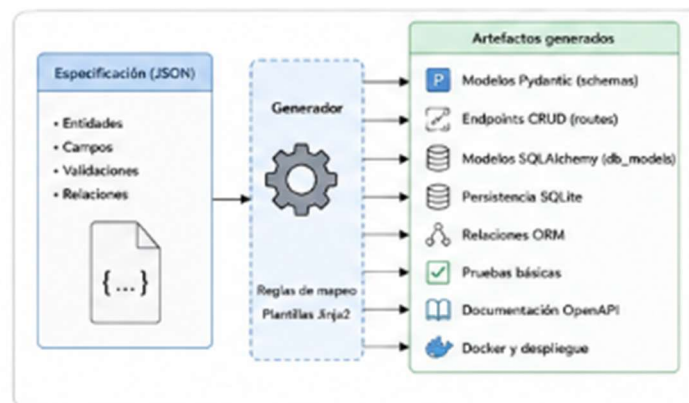


Ilustración 5. Relación entre la especificación y los artefactos generados.

Asimismo, el sistema incorpora soporte para persistencia mediante SQLite y SQLAlchemy, permitiendo almacenar información de forma permanente y gestionar relaciones entre entidades mediante mecanismos ORM. Estas capacidades amplían significativamente el alcance funcional de las aplicaciones generadas y las aproximan a escenarios de desarrollo reales.

La incorporación de pruebas automatizadas y artefactos Docker complementa la arquitectura propuesta, proporcionando mecanismos orientados a mejorar la calidad, reproducibilidad y facilidad de despliegue de las soluciones obtenidas.

En conjunto, la arquitectura propuesta demuestra que es posible automatizar una parte significativa del desarrollo backend utilizando especificaciones de alto nivel y técnicas de generación automática de código.

Una vez definido el diseño conceptual y arquitectónico del sistema, el siguiente capítulo describe la implementación realizada, detallando los principales componentes desarrollados y las decisiones técnicas adoptadas durante la construcción del prototipo.

5. Implementación de la solución

Una vez definido el diseño arquitectónico de la propuesta, el siguiente paso consistió en implementar los distintos componentes necesarios para materializar el sistema de generación automática planteado.

La implementación se desarrolló utilizando Python como lenguaje principal y se estructuró siguiendo una arquitectura modular que permitiera separar claramente las responsabilidades asociadas a la lectura de especificaciones, la generación de código y la producción de artefactos auxiliares.

El desarrollo se realizó de forma incremental, incorporando progresivamente nuevas funcionalidades sobre una base inicial centrada en la generación automática de modelos y endpoints REST. A medida que el proyecto evolucionó, se añadieron capacidades relacionadas con validaciones, persistencia de datos, relaciones entre entidades, pruebas automatizadas y mecanismos de despliegue.

Como resultado, se obtuvo un sistema capaz de generar aplicaciones backend funcionales a partir de especificaciones estructuradas definidas en formato JSON, reduciendo significativamente la cantidad de código que debe implementarse manualmente.

En este capítulo se describen los principales componentes desarrollados, las decisiones técnicas adoptadas durante la implementación y la evolución seguida por el prototipo hasta alcanzar su versión final.

5.1 Estructura general del proyecto

La implementación del sistema se organizó siguiendo una estructura modular cuyo objetivo principal consistió en separar las diferentes responsabilidades asociadas al proceso de generación automática.

Esta organización permitió mantener una arquitectura clara y fácilmente extensible, facilitando tanto el desarrollo incremental como la incorporación de nuevas funcionalidades durante la evolución del proyecto.

El sistema se compone de varios módulos especializados encargados de gestionar la lectura de especificaciones, la generación de código y la producción de artefactos auxiliares.

El directorio *input* contiene las especificaciones JSON utilizadas como entrada del sistema.

El módulo *parser.py* es responsable de interpretar dichas especificaciones y transformarlas en estructuras de datos utilizables por el motor de generación.

Por su parte, el archivo *codegen.py* constituye el núcleo de la aplicación y coordina todo el proceso de generación automática.

Las plantillas utilizadas para producir los distintos componentes software se almacenan en el directorio *templates*, mientras que el resultado final del proceso se genera dentro del directorio *output/generated_api*.

Esta organización modular permitió mantener una clara separación entre especificaciones, lógica de generación y artefactos producidos, facilitando la mantenibilidad y evolución futura del sistema.

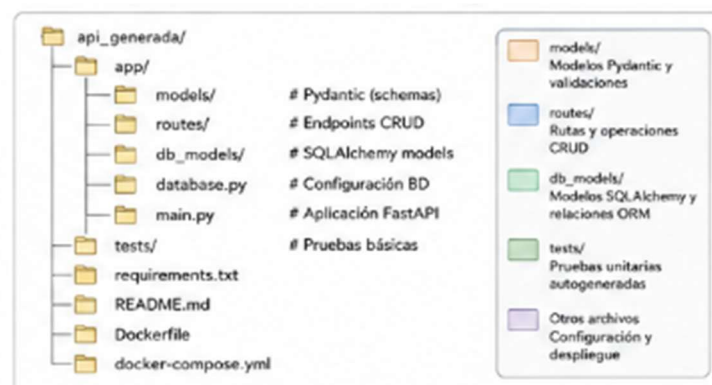


Ilustración 6. Estructura de la aplicación generada.

5.2 Implementación del parser

El parser constituye el primer componente ejecutado dentro del proceso de generación automática. Su responsabilidad principal consiste en leer la especificación proporcionada por el usuario, interpretar su contenido y transformarlo en una estructura de datos utilizable por el resto de módulos del sistema.

La implementación desarrollada se realizó utilizando las capacidades nativas de procesamiento JSON disponibles en Python, evitando dependencias externas innecesarias y manteniendo la simplicidad de la solución.

El componente se implementó dentro del archivo *parser.py*, cuya función principal consiste en cargar la especificación de entrada y devolver una representación estructurada basada en diccionarios y listas de Python.

La simplicidad de esta implementación es posible gracias a la utilización de JSON como formato de entrada, ya que existe una correspondencia directa entre los elementos definidos en la especificación y las estructuras de datos proporcionadas por Python.

Durante la ejecución, el parser recibe la ruta del fichero de especificación y realiza las siguientes operaciones:

1. Apertura del fichero JSON.
2. Lectura del contenido.
3. Conversión a estructuras Python.
4. Devolución de la especificación procesada.

Como resultado de este proceso, el sistema obtiene acceso a toda la información necesaria para iniciar la generación automática de código.

es transformada automáticamente en una estructura de datos interna accesible desde el resto de componentes del sistema.

Además de la lectura de la especificación, durante las fases posteriores del desarrollo se incorporaron mecanismos complementarios de procesamiento dentro del módulo de generación. Estos mecanismos permiten enriquecer la información contenida en las entidades antes de ser utilizada por las plantillas, facilitando la conversión de tipos y la adaptación al código Python generado.

La implementación del parser se mantuvo deliberadamente sencilla con el objetivo de favorecer la extensibilidad futura del sistema. Esta decisión permite incorporar posteriormente validaciones más avanzadas, comprobaciones de consistencia o soporte para formatos alternativos sin modificar significativamente la arquitectura global de la solución.

De este modo, el parser actúa como punto de entrada del proceso de generación y constituye la base sobre la que se construyen los componentes descritos en los apartados siguientes.

5.3 Implementación del motor de generación

El motor de generación constituye el componente central de la solución desarrollada. Su responsabilidad consiste en coordinar el proceso completo de transformación de una especificación estructurada en una aplicación backend funcional.

La implementación de este componente se realizó dentro del archivo *codegen.py*, que actúa como punto de entrada principal del sistema y orquesta todas las etapas del proceso de generación.

Desde el punto de vista arquitectónico, el motor de generación se encarga de:

- Cargar la especificación procesada;
- Preparar la información necesaria para las plantillas;
- Generar los distintos artefactos software;
- Crear la estructura de directorios de salida;
- Coordinar la generación de todos los componentes de la aplicación.

La ejecución comienza mediante la carga de la especificación JSON utilizando el parser descrito en el apartado anterior.

Una vez obtenida la especificación, el sistema invoca secuencialmente los distintos módulos de generación implementados.

Cada una de estas funciones es responsable de generar un tipo específico de artefacto dentro de la aplicación resultante.

Para facilitar la reutilización de código, se implementó una función encargada de inicializar el entorno Jinja2 utilizado por todas las etapas de generación.

Esta aproximación permite centralizar la configuración del motor de plantillas y garantizar un comportamiento homogéneo durante todo el proceso.

Asimismo, antes de generar los distintos archivos, el sistema realiza una fase de enriquecimiento de la información obtenida desde la especificación. Esta

transformación añade información adicional que simplifica la generación posterior de código.

Por ejemplo, los tipos definidos dentro de la especificación son convertidos automáticamente a tipos compatibles con Python.

Este mecanismo permite que las plantillas accedan directamente a los tipos adecuados sin necesidad de incorporar lógica compleja de transformación.

Una característica relevante de la implementación es su diseño modular. Cada tipo de artefacto se genera mediante funciones independientes, lo que facilita la incorporación de nuevas capacidades sin afectar significativamente al resto del sistema.

Gracias a esta arquitectura, durante la evolución del proyecto fue posible añadir progresivamente nuevas funcionalidades como modelos SQLAlchemy, pruebas automatizadas y soporte Docker manteniendo inalterado el núcleo principal del generador.

La implementación final del motor de generación permitió construir una solución flexible, extensible y capaz de producir automáticamente aplicaciones backend completas a partir de una única especificación de entrada.

5.4 Implementación de las plantillas

El proceso de generación automática desarrollado en este trabajo se basa en la utilización de plantillas Jinja2 capaces de transformar la información contenida en la especificación en código fuente ejecutable.

Las plantillas constituyen el núcleo del mecanismo de generación, ya que encapsulan la estructura de los distintos artefactos software producidos por el sistema. Gracias a este enfoque, la lógica de generación permanece separada de los datos específicos de cada aplicación, favoreciendo la reutilización y el mantenimiento del código.

Todas las plantillas implementadas se almacenan dentro del directorio *templates* y son procesadas dinámicamente durante la ejecución del generador.

Cada plantilla se encarga de generar un tipo concreto de componente dentro de la aplicación resultante.

Generación de modelos Pydantic

La plantilla *model.j2* produce automáticamente los modelos de validación utilizados por FastAPI y Pydantic.

Durante el proceso de generación, la plantilla recorre los atributos definidos en cada entidad y genera las anotaciones de tipo correspondientes.

A partir de esta definición, el sistema genera automáticamente modelos adaptados a las características de cada entidad.

Generación de routers FastAPI

La plantilla *router.j2* es responsable de producir los endpoints REST asociados a cada entidad.

Mediante el uso de variables y estructuras de control de Jinja2, se generan automáticamente las operaciones CRUD necesarias para gestionar los recursos definidos en la especificación.

Este mecanismo evita la repetición de código y garantiza una estructura homogénea para todos los routers producidos.

Generación de modelos SQLAlchemy

La plantilla *sqlalchemy_model.j2* genera los modelos ORM utilizados para la persistencia de datos.

Además de los atributos básicos, esta plantilla incorpora automáticamente claves primarias, claves foráneas y relaciones ORM cuando dichas características aparecen definidas en la especificación.

Gracias a esta funcionalidad, el sistema es capaz de producir estructuras de persistencia complejas sin intervención manual.

Generación de configuración y archivos auxiliares

Las plantillas *main.j2* y *database.j2* generan los componentes necesarios para inicializar la aplicación y configurar el acceso a la base de datos.

Asimismo, las plantillas *test_router.j2*, *requirements.j2* y *Dockerfile.j2* producen automáticamente pruebas básicas, dependencias de ejecución y configuraciones de despliegue.

Ventajas del enfoque basado en plantillas

La utilización de Jinja2 proporciona diversas ventajas dentro del sistema desarrollado.

En primer lugar, permite separar claramente la lógica de generación de la información específica de cada proyecto. En segundo lugar, facilita la incorporación de nuevas funcionalidades mediante la creación de plantillas adicionales. Finalmente, mejora la mantenibilidad del sistema al concentrar la estructura de cada artefacto en un único punto de definición.

Durante el desarrollo del proyecto, esta arquitectura permitió incorporar progresivamente nuevas capacidades al generador sin necesidad de modificar significativamente el resto de componentes del sistema.

Como resultado, el motor de generación es capaz de producir aplicaciones backend completas reutilizando un conjunto reducido de plantillas parametrizadas y manteniendo una elevada consistencia entre todos los artefactos generados.

5.5 Implementación de la persistencia

Durante las primeras fases del desarrollo, las aplicaciones generadas utilizaban estructuras de datos temporales para almacenar la información gestionada por los endpoints REST. Aunque este enfoque permitía validar el funcionamiento básico del sistema, presentaba una limitación fundamental: los datos desaparecían al finalizar la ejecución de la aplicación.

Con el objetivo de aproximar la solución a escenarios reales de desarrollo backend, se incorporó una capa completa de persistencia basada en SQLite y SQLAlchemy. Esta ampliación permitió que las aplicaciones generadas almacenasen información de forma permanente y gestionasen operaciones CRUD sobre una base de datos relacional.

La implementación de esta funcionalidad requirió la generación automática de varios componentes especializados.

Configuración de la base de datos

El primer paso consistió en implementar la generación automática del archivo *database.py*, encargado de centralizar toda la configuración relacionada con el acceso a datos.

Este componente define la conexión con SQLite, la creación del motor SQLAlchemy y la configuración de las sesiones utilizadas por la aplicación.

Esta configuración permite que todas las aplicaciones generadas utilicen una estructura homogénea para gestionar el acceso a la base de datos.

Gestión de sesiones

Para facilitar la integración con FastAPI, se implementó un mecanismo de inyección de dependencias basado en sesiones SQLAlchemy.

La función generada proporciona una sesión independiente para cada petición y garantiza su correcta liberación una vez finalizada la operación.

Este enfoque sigue las recomendaciones oficiales de FastAPI y SQLAlchemy para la gestión eficiente de recursos.

Creación automática de tablas

Otro aspecto importante de la implementación consistió en automatizar la creación de las tablas necesarias para almacenar las entidades generadas.

Para ello, la aplicación inicializa automáticamente la estructura de la base de datos durante el arranque.

Gracias a este mecanismo, las tablas correspondientes a las entidades definidas en la especificación son creadas sin necesidad de intervención manual.

Evolución de las operaciones CRUD

La incorporación de persistencia obligó a modificar las operaciones CRUD generadas por el sistema.

Inicialmente, las operaciones utilizaban listas Python para almacenar información temporal.

De forma similar, las operaciones de consulta pasaron a utilizar las capacidades de SQLAlchemy para recuperar información persistida.

Resultado de la implementación

La incorporación de persistencia transformó significativamente las capacidades del sistema generado.

Las aplicaciones producidas dejaron de ser simples prototipos funcionales para convertirse en soluciones capaces de:

- Almacenar información de forma permanente;
- Gestionar múltiples entidades;
- Soportar relaciones entre tablas;
- Realizar operaciones CRUD persistentes;
- Trabajar sobre una base de datos real.

Esta evolución constituyó uno de los hitos más relevantes del proyecto, ya que permitió validar que el generador era capaz de producir aplicaciones backend completas y alineadas con las prácticas habituales del desarrollo profesional.

5.6 Implementación de relaciones

Una vez incorporada la persistencia mediante SQLAlchemy, el siguiente paso en la evolución del sistema consistió en proporcionar soporte para relaciones entre entidades.

Las primeras versiones del generador permitían definir entidades independientes, pero no contemplaban mecanismos para representar asociaciones entre diferentes elementos del dominio. Esta limitación reducía considerablemente la capacidad expresiva de las aplicaciones generadas y dificultaba la representación de escenarios reales.

Con el objetivo de superar esta restricción, se amplió tanto la especificación JSON como las plantillas de generación para soportar claves foráneas y relaciones ORM.

Extensión de la especificación

La primera modificación consistió en incorporar información relacional dentro de las entidades definidas en el documento JSON.

De forma similar se añadieron referencias hacia la entidad *Publisher*.

Esta información permitió describir explícitamente las dependencias existentes entre los distintos elementos del dominio.

Generación automática de claves foráneas

Una vez disponible la información relacional dentro de la especificación, se modificó la plantilla encargada de generar los modelos SQLAlchemy.

Durante el proceso de generación, el sistema identifica automáticamente los campos definidos como claves foráneas y genera las restricciones correspondientes.

Gracias a esta transformación automática, las relaciones definidas a nivel conceptual quedan reflejadas directamente en la estructura de la base de datos.

Generación de relaciones ORM

Además de las claves foráneas, el sistema genera automáticamente relaciones ORM utilizando los mecanismos proporcionados por SQLAlchemy.

Estas asociaciones permiten acceder a objetos relacionados sin necesidad de realizar consultas manuales adicionales.

De forma equivalente se generan las relaciones necesarias para el resto de entidades referenciadas.

La incorporación de estas asociaciones simplifica considerablemente la manipulación de los datos y mejora la legibilidad del código generado.

Adaptación de las respuestas de la API

Una vez implementadas las relaciones ORM, se realizaron modificaciones adicionales para permitir que los endpoints devolvieran información relacional enriquecida.

Inicialmente, las respuestas contenían únicamente los identificadores asociados

Posteriormente, el sistema evolucionó para devolver también los objetos relacionados.

Esta mejora permitió construir respuestas significativamente más útiles para los consumidores de la API.

Validación de la funcionalidad

La implementación de relaciones fue validada mediante pruebas realizadas sobre el caso de estudio desarrollado durante el proyecto.

Se comprobó el correcto funcionamiento de:

- Claves foráneas;
- Asociaciones ORM;
- Recuperación de entidades relacionadas;
- Navegación relacional;
- Respuestas enriquecidas.

Los resultados obtenidos confirmaron que el sistema era capaz de generar automáticamente modelos de dominio con relaciones complejas manteniendo la coherencia entre la especificación de entrada, la estructura de la base de datos y los endpoints expuestos por la API.

La incorporación de soporte para relaciones representa uno de los avances más significativos realizados durante la evolución del prototipo, ya que permitió pasar de aplicaciones basadas en entidades aisladas a sistemas capaces de representar dominios de información más próximos a escenarios reales de desarrollo.

5.7 Implementación de pruebas automáticas

Con el objetivo de incorporar mecanismos básicos de aseguramiento de calidad dentro de las aplicaciones generadas, se desarrolló un sistema capaz de producir automáticamente pruebas utilizando Pytest.

La incorporación de pruebas automatizadas permitió verificar el correcto funcionamiento de los endpoints generados y facilitó la detección temprana de errores durante la evolución del proyecto.

Generación automática de pruebas

Para implementar esta funcionalidad se creó una nueva plantilla denominada *test_router.j2*, encargada de producir automáticamente un conjunto de pruebas para cada entidad definida en la especificación.

Durante el proceso de generación, el sistema recorre las entidades disponibles y crea un archivo de pruebas independiente para cada una de ellas.

estructura resultante es similar a la siguiente:

Cada archivo contiene pruebas básicas orientadas a validar el comportamiento de los endpoints asociados a la entidad correspondiente.

Integración con FastAPI TestClient

Las pruebas generadas utilizan *TestClient*, componente proporcionado por FastAPI para simular peticiones HTTP sobre la aplicación sin necesidad de desplegar un servidor real.

Este enfoque permite verificar automáticamente el comportamiento de los endpoints y detectar posibles errores de implementación.

Problemas detectados durante la validación

Durante las primeras ejecuciones de las pruebas aparecieron errores relacionados con la persistencia de datos.

Concretamente, varias pruebas producían excepciones de integridad debido a la reutilización de identificadores ya existentes en la base de datos SQLite utilizada durante las ejecuciones anteriores.

Los errores obtenidos eran similares al siguiente:

Situaciones equivalentes se detectaron también para las entidades *Author* y *Publisher*.

Estos problemas no estaban relacionados con la generación de código en sí misma, sino con la reutilización de datos persistidos entre diferentes ejecuciones de las pruebas.

Resolución de incidencias

La validación se completó reiniciando el estado de la base de datos antes de ejecutar nuevamente las pruebas.

Una vez corregida esta situación, todas las pruebas generadas se ejecutaron correctamente y verificaron el funcionamiento esperado de los endpoints.

Resultados obtenidos

Tras la resolución de las incidencias detectadas, el conjunto completo de pruebas fue ejecutado satisfactoriamente.

Este resultado confirmó el correcto funcionamiento de:

- Operaciones CRUD generadas;
- Integración con SQLite;
- Modelos SQLAlchemy;
- Relaciones entre entidades;
- Configuración general de la aplicación.

Valoración de la funcionalidad

La generación automática de pruebas representa una mejora significativa respecto a versiones anteriores del sistema, ya que introduce mecanismos básicos de validación sin requerir intervención manual por parte del desarrollador.

Además de facilitar la detección de errores, esta funcionalidad demuestra que el generador es capaz de producir no solo aplicaciones funcionales, sino también artefactos orientados al aseguramiento de calidad del software.

La integración de Pytest contribuye a aproximar las aplicaciones generadas a prácticas habituales del desarrollo profesional y refuerza la fiabilidad de las soluciones producidas por el sistema.

5.8 Implementación del soporte Docker

Como complemento a las funcionalidades principales del sistema, se incorporó soporte para la generación automática de artefactos orientados al despliegue de las aplicaciones producidas.

El objetivo de esta funcionalidad consistió en facilitar la ejecución de las APIs generadas en entornos diferentes al utilizado durante el desarrollo, garantizando además la reproducibilidad del entorno de ejecución.

Para ello, se implementaron nuevas plantillas encargadas de generar automáticamente los ficheros necesarios para la construcción de contenedores Docker.

Generación de dependencias

La primera incorporación consistió en la generación automática del fichero *requirements.txt*, responsable de recoger las dependencias necesarias para ejecutar la aplicación generada.

Esta funcionalidad se implementó mediante una plantilla específica denominada *requirements.j2*.

Gracias a este archivo, cualquier usuario puede instalar automáticamente las bibliotecas necesarias utilizando la herramienta *pip*.

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

Generación automática del Dockerfile

Posteriormente, se desarrolló una nueva plantilla denominada *Dockerfile.j2* encargada de producir automáticamente el fichero Dockerfile asociado a cada aplicación generada.

Este archivo contiene todas las instrucciones necesarias para construir una imagen Docker capaz de ejecutar la aplicación generada.

Integración dentro del generador

La generación de estos artefactos fue integrada dentro del flujo principal del sistema mediante una función específica encargada de procesar ambas plantillas.

De este modo, cada vez que se ejecuta el generador, la aplicación resultante incorpora automáticamente todos los elementos necesarios para su despliegue.

Validación de la funcionalidad

Durante la fase final del proyecto se verificó la correcta generación de los ficheros producidos.

Asimismo, se comprobó que las imágenes Docker podían construirse correctamente utilizando los comandos estándar proporcionados por la plataforma.

Resultado obtenido

La incorporación de soporte Docker permitió ampliar el alcance del sistema más allá de la simple generación de código fuente.

Las aplicaciones producidas pueden distribuirse y ejecutarse de forma consistente en diferentes entornos, reduciendo problemas relacionados con configuraciones locales o dependencias incompatibles.

Aunque esta funcionalidad no constituye el núcleo principal del trabajo, representa un complemento valioso que acerca las aplicaciones generadas a prácticas habituales del desarrollo y despliegue de software moderno.

Con esta incorporación, el generador es capaz de producir no solo el código necesario para implementar una API REST, sino también los artefactos auxiliares requeridos para su instalación y ejecución.

5.9 Evolución del prototipo

El desarrollo del sistema se llevó a cabo de forma incremental, incorporando progresivamente nuevas capacidades sobre una arquitectura inicial sencilla. Este enfoque permitió validar cada funcionalidad de manera independiente antes de integrarla en el conjunto de la solución.

La evolución seguida durante el proyecto puede dividirse en varias versiones funcionales, cada una de ellas orientada a resolver una limitación identificada en la etapa anterior.

Versión 1: Interpretación de especificaciones JSON

La primera versión del prototipo se centró exclusivamente en la lectura e interpretación de especificaciones definidas en formato JSON.

Durante esta etapa se implementó el parser encargado de cargar las entidades y atributos descritos por el usuario, proporcionando una representación interna utilizable por el resto del sistema.

Esta versión permitió validar la viabilidad del formato de entrada y establecer las bases para el proceso de generación automática.

Versión 2: Generación de modelos Pydantic

Una vez completada la interpretación de especificaciones, se incorporó la generación automática de modelos Pydantic.

Esta funcionalidad permitió transformar las entidades descritas en el documento JSON en clases Python capaces de validar automáticamente los datos recibidos por la API.

Además, se añadieron mecanismos para convertir los tipos definidos en la especificación a tipos compatibles con Python.

Versión 3: Generación de routers CRUD

La siguiente evolución consistió en generar automáticamente endpoints REST utilizando FastAPI.

Durante esta fase se implementaron las operaciones CRUD básicas necesarias para gestionar las entidades definidas en la especificación.

Inicialmente, los datos eran almacenados en estructuras temporales en memoria, lo que permitió validar el funcionamiento general de los endpoints sin incorporar todavía mecanismos de persistencia.

Versión 4: Incorporación de persistencia SQLite

Tras validar el funcionamiento básico de las operaciones CRUD, se identificó la necesidad de almacenar la información de forma permanente.

Para resolver esta limitación se incorporó soporte para SQLite y se generó automáticamente la configuración necesaria para gestionar la conexión con la base de datos.

Esta evolución permitió conservar los datos entre ejecuciones y aproximar el sistema a escenarios reales de utilización.

Versión 5: Integración de SQLAlchemy ORM

Posteriormente, se añadió soporte para SQLAlchemy como mecanismo ORM.

Esta incorporación permitió gestionar entidades persistentes mediante objetos Python y simplificó significativamente la implementación de operaciones CRUD sobre la base de datos.

Además, se generaron automáticamente modelos ORM independientes para cada entidad definida en la especificación.

Versión 6: Soporte para relaciones entre entidades

Una de las ampliaciones más relevantes del proyecto consistió en incorporar soporte para relaciones entre entidades.

Para ello, se extendió la especificación JSON y se modificaron las plantillas de generación con el fin de producir claves foráneas, asociaciones ORM y respuestas relacionales enriquecidas.

Gracias a esta evolución, el sistema pasó de gestionar entidades independientes a representar modelos de dominio más complejos y próximos a escenarios reales.

Versión 7: Generación automática de pruebas

Con el objetivo de mejorar la calidad del software generado, se implementó un mecanismo capaz de producir automáticamente pruebas utilizando Pytest.

Esta funcionalidad permitió validar el correcto funcionamiento de los endpoints generados y facilitó la detección temprana de errores durante la evolución del sistema.

La ejecución satisfactoria de las pruebas confirmó la estabilidad de las funcionalidades incorporadas.

Versión 8: Soporte para despliegue mediante Docker

Finalmente, se añadieron mecanismos orientados a facilitar la distribución y despliegue de las aplicaciones generadas.

Durante esta fase se implementó la generación automática de ficheros *requirements.txt* y *Dockerfile*, permitiendo construir imágenes Docker listas para su ejecución.

Esta funcionalidad completó el ciclo de generación, proporcionando no solo el código fuente de la aplicación, sino también los elementos necesarios para su despliegue.

Resultado final

Como resultado de esta evolución incremental, se obtuvo un sistema capaz de transformar una especificación estructurada en una aplicación backend completa, incluyendo validación, persistencia, relaciones entre entidades, pruebas automatizadas y soporte para despliegue.

La progresión seguida durante el proyecto permitió validar cada funcionalidad de manera independiente y construir gradualmente una solución más robusta y funcional, demostrando la viabilidad de la generación automática de APIs REST a partir de especificaciones de alto nivel.

6. Evaluación experimental

Una vez completado el desarrollo del sistema, resulta necesario evaluar su funcionamiento y analizar el impacto real de la generación automática sobre el proceso de construcción de aplicaciones backend.

El objetivo de esta evaluación consiste en determinar si la solución propuesta permite reducir el esfuerzo de desarrollo manteniendo al mismo tiempo la funcionalidad y calidad de las aplicaciones obtenidas.

Para ello se diseñó un caso de estudio basado en una aplicación de gestión bibliográfica, utilizada como escenario experimental para comparar el desarrollo manual de una API REST frente al enfoque basado en generación automática implementado en este trabajo.

La evaluación se centra principalmente en métricas relacionadas con productividad, volumen de código generado, tiempo de desarrollo y nivel de automatización alcanzado.

Los apartados siguientes describen el caso de estudio utilizado, la metodología experimental aplicada y los resultados obtenidos durante la validación de la propuesta.

6.1 Objetivos de la evaluación

La evaluación experimental realizada persigue analizar la utilidad práctica del sistema desarrollado y cuantificar los beneficios obtenidos mediante la generación automática de código.

Concretamente, se pretende responder a las siguientes cuestiones de investigación:

Q1. ¿Reduce la generación automática el tiempo necesario para desarrollar una API REST?

Se analiza el tiempo requerido para construir una aplicación backend utilizando el generador y se compara con una implementación manual equivalente.

Q2. ¿Disminuye la cantidad de código que debe escribir el desarrollador?

Se estudia la diferencia entre el volumen de código implementado manualmente y la información que debe proporcionar el usuario mediante la especificación JSON.

Q3. ¿Es capaz el sistema de generar aplicaciones funcionales?

Se verifica que las aplicaciones producidas incorporan correctamente modelos, operaciones CRUD, persistencia, relaciones y pruebas automatizadas.

Q4. ¿Puede considerarse viable el enfoque propuesto?

Se analiza si los resultados obtenidos justifican la utilización de técnicas de generación automática para acelerar el desarrollo backend.

Para responder a estas cuestiones se diseñó un caso de estudio representativo que permitió comparar ambos enfoques bajo condiciones similares.

6.2 Caso de estudio

Para evaluar la propuesta desarrollada se seleccionó como caso de estudio una aplicación de gestión bibliográfica implementada mediante una API REST.

La elección de este escenario se realizó debido a que presenta características habituales en aplicaciones backend reales, incluyendo múltiples entidades, relaciones entre datos, operaciones CRUD y persistencia en base de datos.

La aplicación desarrollada se estructura en torno a tres entidades principales:

- **Book**, utilizada para representar libros.
- **Author**, utilizada para representar autores.
- **Publisher**, utilizada para representar editoriales.

Estas entidades permiten modelar un dominio suficientemente representativo para validar las capacidades incorporadas al generador.

La entidad *Book* incluye información descriptiva de cada libro y mantiene relaciones con las entidades *Author* y *Publisher* mediante claves foráneas.

La Figura muestra una representación simplificada del modelo utilizado.

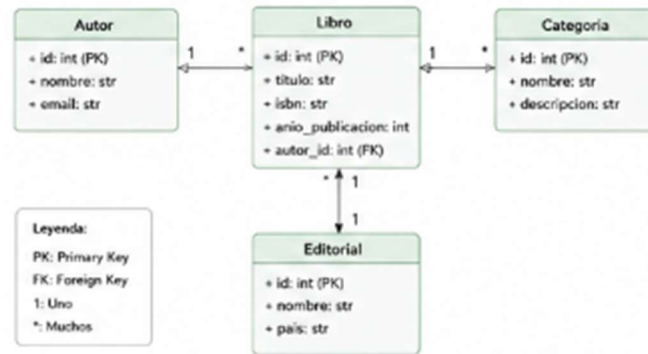


Ilustración 7. Diagrama de clases del caso de estudio.

La especificación utilizada como entrada del sistema fue definida mediante un documento JSON que describe las entidades, atributos, validaciones y relaciones necesarias para construir la aplicación.

A partir de esta especificación, el generador produjo automáticamente todos los componentes software necesarios para obtener una API funcional.

Entre los artefactos generados se encuentran:

- Modelos Pydantic;
- Routers FastAPI;
- Modelos SQLAlchemy;
- Configuración de persistencia;
- Pruebas automatizadas;
- Documentación OpenAPI;
- Artefactos Docker.

La aplicación generada permite realizar operaciones CRUD completas sobre las tres entidades definidas, almacenar información de forma persistente en SQLite y navegar entre entidades relacionadas mediante mecanismos ORM.

Además, la API incorpora documentación automática accesible mediante Swagger UI, facilitando la validación funcional de los endpoints generados.

Este caso de estudio fue utilizado como base para comparar el desarrollo manual de una aplicación equivalente frente al enfoque basado en generación automática propuesto en este trabajo.

Los resultados obtenidos permiten analizar el impacto de la automatización sobre distintos aspectos del proceso de desarrollo, incluyendo productividad, volumen de código y esfuerzo requerido para construir una aplicación backend funcional.

6.3 Metodología experimental

Con el objetivo de evaluar la eficacia de la solución propuesta, se diseñó una metodología experimental basada en la comparación entre el desarrollo manual de una aplicación backend y la generación automática de una aplicación equivalente utilizando el sistema desarrollado.

La evaluación se realizó utilizando el caso de estudio descrito en el apartado anterior, correspondiente a una aplicación de gestión bibliográfica compuesta por las entidades *Book*, *Author* y *Publisher*.

Ambos enfoques persiguieron obtener una aplicación con características funcionales equivalentes, incluyendo:

- Definición de modelos de datos;
- Operaciones CRUD;
- Persistencia mediante SQLite;
- Integración con SQLAlchemy;
- Relaciones entre entidades;
- Pruebas automatizadas;
- Soporte para despliegue.

Desarrollo manual

En el primer escenario se consideró el desarrollo tradicional de una aplicación backend utilizando FastAPI.

Este proceso requiere implementar manualmente todos los componentes necesarios para la aplicación, incluyendo modelos, routers, configuración de persistencia, relaciones ORM, pruebas y artefactos de despliegue.

Cada modificación o ampliación del dominio implica además actualizar múltiples archivos de forma coordinada.

Generación automática

En el segundo escenario se utilizó el sistema desarrollado en este trabajo.

El desarrollador únicamente proporciona una especificación JSON describiendo las entidades y relaciones del dominio. A partir de esta información, el generador produce automáticamente todos los componentes software necesarios para construir la aplicación.

La intervención manual queda reducida a la definición de la especificación y a la ejecución del proceso de generación.

Métricas analizadas

Para realizar la comparación se definieron varias métricas orientadas a evaluar el impacto de la automatización sobre el proceso de desarrollo.

Las métricas consideradas fueron las siguientes:

Tiempo de desarrollo

Se analizó el tiempo necesario para obtener una aplicación funcional en ambos escenarios.

En el caso manual, este tiempo incluye la implementación de todos los componentes necesarios. En el caso automático, únicamente se considera la elaboración de la especificación y la ejecución del generador.

Volumen de código

Se comparó la cantidad de código que debe producir el desarrollador en cada enfoque.

Mientras que el desarrollo tradicional requiere implementar múltiples archivos fuente, la generación automática se basa en una única especificación de entrada significativamente más compacta.

Número de artefactos generados

Se evaluó la cantidad de componentes software obtenidos automáticamente por el sistema.

Esta métrica permite cuantificar el nivel de automatización alcanzado y el alcance funcional de la solución propuesta.

Funcionalidad obtenida

Finalmente, se verificó que las aplicaciones generadas incorporaban correctamente las funcionalidades previstas, incluyendo persistencia, relaciones entre entidades, pruebas automatizadas y mecanismos de despliegue.

Procedimiento experimental

La evaluación se llevó a cabo siguiendo las siguientes etapas:

1. Definición de la especificación del caso de estudio.
2. Generación automática de la aplicación backend.
3. Ejecución de pruebas funcionales y automatizadas.
4. Comparación con una implementación manual equivalente.
5. Análisis de las métricas obtenidas.

Los resultados obtenidos mediante esta metodología se presentan y analizan en los apartados siguientes.

6.4 Resultados obtenidos

La aplicación generada a partir de la especificación del caso de estudio fue construida correctamente y ejecutada sin necesidad de modificaciones manuales sobre el código producido.

El sistema generó automáticamente todos los componentes necesarios para obtener una API backend funcional, incluyendo modelos de validación, endpoints REST, persistencia de datos, relaciones entre entidades, pruebas automatizadas y artefactos de despliegue.

Resultados funcionales

Durante la validación se comprobó el correcto funcionamiento de las principales capacidades incorporadas por el generador.

La Tabla resume los resultados obtenidos.

Tabla 2. Resumen de los resultados obtenidos.

Funcionalidad	Resultado
Generación de modelos Pydantic	Correcto
Generación de routers FastAPI	Correcto
Persistencia SQLite	Correcto
Modelos SQLAlchemy	Correcto
Relaciones entre entidades	Correcto
Navegación relacional	Correcto
Generación de pruebas	Correcto
Generación de Dockerfile	Correcto

Los resultados obtenidos indican que el sistema fue capaz de generar automáticamente una aplicación backend completa manteniendo la coherencia entre todos los componentes producidos.

Resultados de las pruebas automatizadas

Las pruebas generadas automáticamente mediante Pytest fueron ejecutadas sobre la aplicación producida.

Tras resolver las incidencias iniciales relacionadas con la reutilización de datos persistidos en SQLite, todas las pruebas finalizaron satisfactoriamente.

Este resultado confirma el correcto funcionamiento de los endpoints generados y la adecuada integración entre FastAPI, SQLAlchemy y SQLite.

Comparación preliminar con el desarrollo manual

La comparación entre el enfoque tradicional y la generación automática muestra una reducción significativa del esfuerzo requerido para construir una aplicación backend equivalente.

Mientras que en un desarrollo convencional resulta necesario implementar manualmente todos los componentes de la aplicación, el sistema desarrollado permite obtener los mismos artefactos a partir de una única especificación JSON.

La Tabla muestra el esquema de comparación utilizado durante la evaluación.

Tabla 3. Esquema comparativo.

Métrica	Desarrollo manual	Generación automática
Entidades implementadas	3	3
Operaciones CRUD	Sí	Sí
Persistencia	Sí	Sí
Relaciones ORM	Sí	Sí
Pruebas automatizadas	Sí	Sí
Docker	Sí	Sí

Los resultados cuantitativos detallados se analizan en el apartado siguiente, donde se estudia el impacto de la automatización sobre la productividad y el esfuerzo de desarrollo.

6.5 Análisis de resultados

Los resultados obtenidos durante la evaluación permiten extraer diversas conclusiones acerca de la utilidad práctica del sistema desarrollado y del impacto de la generación automática sobre el proceso de construcción de aplicaciones backend.

En primer lugar, la evaluación confirmó que el generador es capaz de producir aplicaciones completamente funcionales a partir de una única especificación estructurada. Los artefactos generados incluyen todos los componentes necesarios para la ejecución de la aplicación, abarcando validación de datos, persistencia, relaciones entre entidades, pruebas automatizadas y mecanismos de despliegue.

Desde el punto de vista funcional, la aplicación generada presentó un comportamiento equivalente al esperado en una implementación manual. Las operaciones CRUD, la persistencia mediante SQLite, las relaciones ORM y las pruebas automatizadas funcionaron correctamente durante las fases de validación.

Impacto sobre la productividad

Uno de los principales objetivos del trabajo consistía en reducir el esfuerzo necesario para desarrollar aplicaciones backend repetitivas.

Los resultados obtenidos muestran que el enfoque basado en generación automática permite sustituir la implementación manual de múltiples componentes por una única especificación JSON de alto nivel.

En un desarrollo convencional, el programador debe implementar y mantener manualmente modelos, routers, configuraciones de persistencia, relaciones ORM, pruebas y artefactos auxiliares. Por el contrario, en la solución propuesta estos componentes son producidos automáticamente por el generador.

Como consecuencia, el esfuerzo de desarrollo se desplaza desde la escritura de código repetitivo hacia la descripción declarativa del dominio de aplicación.

Reducción de tareas repetitivas

Otro aspecto relevante observado durante la evaluación fue la eliminación de gran parte de las tareas repetitivas asociadas al desarrollo backend.

La creación manual de estructuras CRUD para múltiples entidades suele implicar una elevada cantidad de código similar. Esta situación aumenta el tiempo de desarrollo y favorece la aparición de errores derivados de duplicación o inconsistencias entre componentes.

La utilización de plantillas permitió automatizar completamente este proceso, garantizando además una estructura homogénea para todos los artefactos generados.

Consistencia del software generado

La generación automática proporciona un nivel elevado de consistencia entre los distintos componentes de la aplicación.

Dado que todos los artefactos se producen a partir de una única especificación, las entidades, validaciones, relaciones y mecanismos de persistencia permanecen sincronizados automáticamente.

Este aspecto reduce significativamente el riesgo de discrepancias entre modelos, routers y estructuras de almacenamiento, problema frecuente en desarrollos realizados manualmente.

Escalabilidad de la solución

Los resultados obtenidos sugieren que la propuesta puede adaptarse adecuadamente a aplicaciones con un mayor número de entidades.

Una vez definidas las plantillas de generación, la incorporación de nuevas entidades únicamente requiere modificar la especificación de entrada, manteniéndose constante el esfuerzo necesario para producir el resto de componentes.

Esta característica constituye una ventaja importante frente al desarrollo manual, donde el volumen de trabajo crece proporcionalmente al número de elementos del dominio.

Limitaciones identificadas

A pesar de los resultados positivos obtenidos, la evaluación también permitió identificar algunas limitaciones de la solución.

En su estado actual, el sistema se encuentra orientado principalmente a aplicaciones CRUD basadas en entidades y relaciones relativamente sencillas. Escenarios más complejos podrían requerir mecanismos adicionales de personalización o generación avanzada.

Asimismo, determinadas reglas de negocio específicas continúan necesitando intervención manual por parte del desarrollador una vez finalizado el proceso de generación.

Valoración global

Considerando conjuntamente los resultados funcionales y las métricas analizadas, puede concluirse que la propuesta desarrollada cumple satisfactoriamente los objetivos planteados al inicio del trabajo.

El sistema demuestra que es posible automatizar una parte significativa del desarrollo backend mediante técnicas de generación de código basadas en especificaciones declarativas.

La reducción del trabajo manual, la mejora de la consistencia entre componentes y la capacidad para generar aplicaciones funcionales de forma automática constituyen evidencias que respaldan la viabilidad del enfoque propuesto y justifican su interés como línea de investigación y desarrollo dentro del ámbito de la Ingeniería del Software.

6.6 Amenazas a la validez

Como en cualquier trabajo experimental, los resultados obtenidos deben interpretarse teniendo en cuenta determinadas limitaciones que pueden influir en la validez de las conclusiones alcanzadas.

La identificación explícita de estas amenazas permite contextualizar adecuadamente los resultados y proporciona una visión más rigurosa de las capacidades reales de la solución propuesta.

Validez interna

La evaluación se realizó utilizando un único caso de estudio basado en una aplicación de gestión bibliográfica.

Aunque este escenario permitió validar las funcionalidades principales del sistema, los resultados podrían verse influenciados por las características particulares del dominio seleccionado.

La utilización de casos de estudio adicionales pertenecientes a otros ámbitos de aplicación permitiría obtener una evidencia más amplia sobre el comportamiento del generador.

Validez externa

El sistema desarrollado fue evaluado sobre aplicaciones orientadas principalmente a operaciones CRUD y relaciones entre entidades.

Por este motivo, los resultados obtenidos no pueden generalizarse automáticamente a sistemas con lógica de negocio compleja, arquitecturas distribuidas o requisitos funcionales significativamente diferentes.

La capacidad de la solución para adaptarse a este tipo de escenarios requeriría estudios adicionales y la incorporación de nuevas funcionalidades al generador.

Validez de construcción

Las métricas empleadas durante la evaluación se centraron principalmente en aspectos relacionados con productividad, automatización y funcionalidad obtenida.

Aunque estas métricas permiten analizar adecuadamente los objetivos principales del trabajo, existen otros factores relevantes que no fueron estudiados en profundidad, como la mantenibilidad a largo plazo, la complejidad del código generado o el impacto sobre el rendimiento de ejecución.

La inclusión de estas métricas podría proporcionar una visión más completa del comportamiento de la solución.

Validez de conclusión

La comparación realizada entre desarrollo manual y generación automática se basa en una implementación concreta y en un conjunto limitado de escenarios de prueba.

Aunque los resultados muestran una reducción significativa del esfuerzo necesario para construir la aplicación evaluada, las conclusiones deben interpretarse dentro del contexto específico del caso de estudio utilizado.

No obstante, la correcta generación de todos los componentes, el funcionamiento satisfactorio de las pruebas automatizadas y la validación funcional realizada proporcionan evidencia suficiente para considerar viable el enfoque propuesto.

Consideraciones finales

A pesar de las limitaciones identificadas, las amenazas descritas no invalidan los resultados obtenidos, sino que delimitan el alcance de las conclusiones que pueden extraerse a partir de la evaluación realizada.

Los resultados deben entenderse como una validación inicial de la viabilidad de la generación automática de aplicaciones backend a partir de especificaciones estructuradas.

La realización de estudios adicionales sobre dominios más amplios y escenarios de mayor complejidad constituye una línea natural de trabajo futuro que permitiría reforzar y ampliar las conclusiones alcanzadas en este proyecto.

6.7 Resumen de la evaluación

En este capítulo se ha presentado la evaluación experimental realizada sobre el sistema de generación automática desarrollado en este trabajo.

La validación se llevó a cabo utilizando como caso de estudio una aplicación de gestión bibliográfica compuesta por las entidades *Book*, *Author* y *Publisher*, permitiendo analizar el comportamiento del generador en un escenario representativo de desarrollo backend.

Los resultados obtenidos demostraron que el sistema es capaz de producir automáticamente aplicaciones funcionales a partir de una especificación JSON de alto nivel. Entre las capacidades verificadas se encuentran la generación de modelos Pydantic, endpoints REST mediante FastAPI, persistencia utilizando SQLite y SQLAlchemy, gestión de relaciones entre entidades, pruebas automatizadas y soporte para despliegue mediante Docker.

Asimismo, la comparación realizada entre el desarrollo manual y el enfoque basado en generación automática puso de manifiesto una reducción significativa del trabajo repetitivo asociado a la construcción de aplicaciones CRUD. La utilización de plantillas y especificaciones declarativas permitió automatizar gran parte de las tareas necesarias para obtener una aplicación backend completa.

Las pruebas funcionales y automatizadas ejecutadas durante la evaluación confirmaron el correcto funcionamiento de los componentes generados y validaron la coherencia entre los distintos elementos producidos por el sistema.

A pesar de las limitaciones identificadas, los resultados obtenidos proporcionan evidencia suficiente para considerar viable el enfoque propuesto y respaldan la utilidad de las técnicas de generación automática de código en el contexto del desarrollo backend.

En conjunto, la evaluación realizada permite concluir que la solución desarrollada cumple los objetivos planteados al inicio del trabajo y constituye una alternativa válida para reducir el esfuerzo necesario en la construcción de aplicaciones REST basadas en entidades y operaciones CRUD.

Una vez analizados los resultados obtenidos, el capítulo siguiente presenta las conclusiones finales del trabajo, las principales aportaciones realizadas y las posibles líneas de evolución futura de la propuesta desarrollada.

7. Conclusiones y trabajo futuro

Este capítulo presenta las principales conclusiones obtenidas tras el desarrollo y evaluación del sistema propuesto, así como las aportaciones realizadas y las posibles líneas de evolución futura derivadas del trabajo realizado.

A lo largo del proyecto se ha diseñado, implementado y validado una solución orientada a la generación automática de aplicaciones backend a partir de especificaciones estructuradas definidas mediante documentos JSON.

Los resultados obtenidos permiten analizar el grado de cumplimiento de los objetivos planteados inicialmente y valorar el potencial de las técnicas de generación automática dentro del ámbito del desarrollo software.

Asimismo, se identifican posibles mejoras y extensiones que podrían incorporarse en futuras versiones del sistema con el fin de ampliar sus capacidades y aumentar su aplicabilidad en escenarios más complejos.

7.1 Conclusiones

El objetivo principal de este trabajo consistía en investigar la viabilidad de la generación automática de aplicaciones backend a partir de especificaciones de alto nivel y desarrollar un prototipo capaz de materializar dicha aproximación.

Los resultados obtenidos permiten afirmar que este objetivo ha sido alcanzado satisfactoriamente.

A lo largo del proyecto se ha desarrollado un sistema capaz de interpretar especificaciones estructuradas definidas mediante documentos JSON y transformarlas automáticamente en aplicaciones backend funcionales basadas en FastAPI.

La solución implementada genera de forma automática modelos de validación, endpoints REST, componentes de persistencia, modelos ORM, pruebas automatizadas y artefactos de despliegue, reduciendo significativamente la cantidad de código que debe desarrollarse manualmente.

La evaluación realizada confirmó el correcto funcionamiento de las aplicaciones generadas y demostró que el enfoque propuesto permite automatizar una parte importante de las tareas repetitivas asociadas al desarrollo backend.

Asimismo, la incorporación progresiva de funcionalidades como persistencia mediante SQLite, integración con SQLAlchemy, soporte para relaciones entre entidades y generación automática de pruebas permitió evolucionar desde un prototipo básico hacia una solución significativamente más completa y cercana a escenarios reales de desarrollo.

Los resultados obtenidos respaldan la hipótesis inicial de que las técnicas de generación automática pueden contribuir a mejorar la productividad del desarrollo software cuando se aplican a dominios caracterizados por estructuras repetitivas y patrones bien definidos.

En consecuencia, puede concluirse que la propuesta desarrollada constituye una aproximación viable para automatizar la construcción de aplicaciones REST basadas en entidades y operaciones CRUD.

7.2 Aportaciones del trabajo

Las principales aportaciones realizadas durante este trabajo pueden resumirse en los siguientes puntos:

Diseño de una especificación declarativa

Se diseñó un modelo de especificación basado en JSON capaz de describir entidades, atributos, validaciones y relaciones entre elementos del dominio.

Desarrollo de un generador automático de código

Se implementó un motor de generación basado en plantillas Jinja2 capaz de transformar las especificaciones definidas por el usuario en código fuente ejecutable.

Generación automática de componentes backend

El sistema desarrollado genera automáticamente:

- Modelos Pydantic;
- Routers FastAPI;
- Modelos SQLAlchemy;
- Configuración de persistencia;
- Pruebas automatizadas;
- Artefactos Docker.

Incorporación de soporte relacional

Se amplió la solución para gestionar claves foráneas, asociaciones ORM y navegación entre entidades relacionadas.

Validación experimental

Se realizó una evaluación práctica utilizando una aplicación de gestión bibliográfica como caso de estudio, permitiendo verificar la viabilidad de la propuesta y analizar su impacto sobre el proceso de desarrollo.

Contribución al ámbito de la Ingeniería del Software

El trabajo aporta una experiencia práctica sobre la aplicación de técnicas de generación automática de código en el contexto del desarrollo backend, mostrando cómo las especificaciones de alto nivel pueden utilizarse para reducir tareas repetitivas y mejorar la productividad.

8. Bibliografía

- [1] Fowler, M. (2010). Domain-Specific Languages. Addison-Wesley Professional.
- [2] Brambilla, M., Cabot, J., & Wimmer, M. (2017). Model-Driven Software Engineering in Practice (2nd ed.). Morgan & Claypool Publishers.
- [3] Schmidt, D. C. (2006). Model-Driven Engineering. IEEE Computer, 39(2), 25–31.
- [4] Kelly, S., & Tolvanen, J. P. (2008). Domain-Specific Modeling: Enabling Full Code Generation. Wiley-IEEE Computer Society Press.
- [5] Pressman, R. S., & Maxim, B. R. (2020). Software Engineering: A Practitioner's Approach (9th ed.). McGraw-Hill Education.
- [6] Sommerville, I. (2016). Software Engineering (10th ed.). Pearson.
- [7] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
- [8] FastAPI Documentation. FastAPI Framework Documentation. Disponible en:
<https://fastapi.tiangolo.com>
- [9] Pydantic Documentation. Data Validation and Settings Management using Python Type Hints. Disponible en:
<https://docs.pydantic.dev>
- [10] SQLAlchemy Documentation. SQLAlchemy ORM Documentation. Disponible en:
<https://www.sqlalchemy.org>
- [11] SQLite Documentation. SQLite Official Documentation. Disponible en:
<https://www.sqlite.org/docs.html>
- [12] Jinja Documentation. Jinja Template Engine Documentation. Disponible en:
<https://jinja.palletsprojects.com>
- [13] Docker Documentation. Docker Official Documentation. Disponible en:
<https://docs.docker.com>
- [14] Pytest Documentation. Pytest Testing Framework Documentation. Disponible en:
<https://docs.pytest.org>

- [15] Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. Doctoral Dissertation, University of California, Irvine.
- [16] Richardson, L., & Ruby, S. (2007). RESTful Web Services. O'Reilly Media.
- [17] Bass, L., Clements, P., & Kazman, R. (2021). Software Architecture in Practice (4th ed.). Addison-Wesley.
- [18] Kleppe, A., Warmer, J., & Bast, W. (2003). MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley.
- [19] France, R., & Rumpe, B. (2007). Model-Driven Development of Complex Software: A Research Roadmap. Future of Software Engineering, IEEE Computer Society, 37–54.
- [20] Mell, P., & Grance, T. (2011). The NIST Definition of Cloud Computing. National Institute of Standards and Technology.
- [21] Pallets, *Flask Documentation*. Pallets Projects. Disponible en: <https://flask.palletsprojects.com/en/stable/>. Consultado en 2026.
- [22] Django Software Foundation, *Django Documentation*. Disponible en: <https://docs.djangoproject.com/>. Consultado en 2026.
- [23] OpenJS Foundation, *Express.js Documentation*. Disponible en: <https://expressjs.com/>. Consultado en 2026.
- [24] Broadcom, *Spring Boot Reference Documentation*. Disponible en: <https://docs.spring.io/spring-boot/>. Consultado en 2026.
- [25] OpenAPI Generator Contributors, *OpenAPI Generator Documentation*. Disponible en: <https://openapi-generator.tech/>. Consultado en 2026.
- [26] SmartBear Software, *Swagger Codegen Documentation*. Disponible en: <https://swagger.io/docs/open-source-tools/swagger-codegen/>. Consultado en 2026.
- [27] Yeoman, *The web's scaffolding tool for modern webapps*. Disponible en: <https://yeoman.io/>. Consultado en 2026.
- [28] A. Sahay, A. Indamutsa, D. Di Ruscio y A. Pierantonio, “Adoption of low-code and no-code development: A systematic literature review and future research agenda”, *Journal of Systems and Software*, vol. 222, art. 112300, 2025, doi: 10.1016/j.jss.2024.112300.

9. Lista de siglas y acrónimos

Sigla Significado

API	Application Programming Interface
CRUD	Create, Read, Update, Delete
ORM	Object Relational Mapping
JSON	JavaScript Object Notation
REST	Representational State Transfer
SQL	Structured Query Language
ASGI	Asynchronous Server Gateway Interface
MDE	Model-Driven Engineering
TFM	Trabajo Fin de Máster
UI	User Interface

10. Anexos

Anexo A. Especificación JSON utilizada

En este anexo se incluye la especificación JSON completa empleada durante el desarrollo y validación del sistema. Esta especificación describe las entidades del dominio, sus atributos, restricciones de validación y relaciones utilizadas para generar automáticamente la aplicación de gestión bibliográfica presentada como caso de estudio.

Se adjunta el contenido completo del fichero:

input/library.json

Anexo B. Plantillas Jinja2 desarrolladas

Este anexo recoge las plantillas Jinja2 implementadas para la generación automática de los distintos componentes software producidos por el sistema.

Las plantillas desarrolladas son:

templates/

└─ model.j2

└─ router.j2

└─ main.j2

└─ database.j2

└─ sqlalchemy_model.j2

└─ test_router.j2

└─ Dockerfile.j2

└─ requirements.j2

Estas plantillas constituyen el núcleo del mecanismo de generación automática y permiten producir aplicaciones backend completas a partir de una especificación declarativa.

Anexo C. Resultados de las pruebas

Con el objetivo de verificar el correcto funcionamiento de la aplicación generada, se ejecutaron las pruebas automáticas producidas por el sistema mediante Pytest.

Los resultados obtenidos fueron satisfactorios para todas las entidades del caso de estudio.

Asimismo, se realizaron pruebas funcionales utilizando Swagger UI para validar:

- Operaciones CRUD;
- Persistencia de datos;
- Relaciones entre entidades;
- Navegación relacional;
- Integración con SQLite;
- Integración con SQLAlchemy.

Los resultados obtenidos confirmaron el correcto funcionamiento de la aplicación generada y la coherencia entre los distintos componentes producidos automáticamente por el sistema.

Anexo D. Dockerfile y dependencias generadas

Como parte de las funcionalidades de despliegue incorporadas al sistema, se generaron automáticamente los siguientes artefactos:

Dockerfile
requirements.txt

Estos archivos permiten reproducir el entorno de ejecución de la aplicación y desplegarla mediante contenedores Docker.

Las dependencias generadas incluyen:

fastapi
uvicorn
pydantic
sqlalchemy
pytest
httpx
jinja2

La generación automática de estos artefactos facilita la instalación, distribución y ejecución de las aplicaciones producidas por el sistema.