



**CIPFP Mislata**

Centre Integrat Públic  
Formació Professional Superior

## **PLATAFORMA EDUCATIVA GOLEARNIX:**

### APRENDE CONSTRUYENDO, CONSTRUYE APRENDIENDO



Javier Fernández Díaz-Guerra

*Proyecto de Desarrollo de Aplicaciones Web*

*Curso 2024-2025*

*Tutor de proyecto: César Guijarro*





**CREATIVE  
COMMONS**

**CC0 1.0  
UNIVERSAL**

**CREATIVE COMMONS LEGAL CODE**

You must retain a clear image of your purpose.

You will never know how good you might have  
become unless you try.

- Mike Mentzer

## **AGRADECIMIENTOS**

Me gustaría agradecer a todas aquellas personas que de alguna forma u otra han participado en este proyecto y han hecho que su realización sea posible.

En primer lugar, agradecer a mi tutor de proyecto, César Guijarro, ya que sin su ayuda, paciencia y comprensión este proyecto no se hubiera podido realizar.

Por último, y no menos importante, agradecer a mi familia, por todo el apoyo y ayuda que me brindan siempre, ya que, sin ellos, también hubiera sido imposible realizar este proyecto.

¡Muchísimas gracias a todos! Sin vosotros no hubiera podido llegar aquí.

## RESUMEN

El presente trabajo se centra en el estudio, diseño e implementación de una plataforma educativa llamada GoLearnix. Esta plataforma actúa como un puente para explorar buenas prácticas de desarrollo web moderno, utilizando arquitecturas avanzadas, aplicando patrones de diseño y manteniendo una clara separación de responsabilidades, especialmente entre los módulos de autenticación y gestión de cursos.

Para llevar a cabo este proyecto, se realizará un análisis detallado de las tecnologías utilizadas, con el objetivo de justificar cada elección técnica. Se trabajará con diversos lenguajes de programación, como Go y Java, junto con sus frameworks más representativos, los cuales se describirán en detalle más adelante.

Asimismo, se explorarán distintas arquitecturas de software, tales como la arquitectura de microservicios, hexagonal y de sacrificio, entre otras. El proyecto también destaca por su enfoque innovador en el uso de eventos y en la aplicación de patrones de diseño avanzados.

## ABSTRACT

This work focuses on the study, design, and implementation of an educational platform named GoLearnix. The platform serves as a bridge to explore best practices in modern web development, employing advanced architectures, applying design patterns, and maintaining a clear separation of concerns, particularly between the authentication and course management modules.

To carry out this project, a thorough analysis of the technologies employed will be conducted, with the aim of justifying each technical decision. A variety of programming languages will be used, including Go and Java, along with their most representative frameworks, which will be described in detail later on.

Furthermore, the project will explore various software architectures, such as microservices, hexagonal, and sacrifice architectures, among others. It also stands out for its innovative approach to event-driven design and the application of advanced design patterns.



## ÍNDICE

<b>1. INTRODUCCIÓN.....</b>	<b>- 10 -</b>
1.1. Entorno y Contextualización .....	- 11 -
1.2. Objetivos .....	- 12 -
<b>2. MARCO CONCEPTUAL.....</b>	<b>- 13 -</b>
<b>3. TEMPORALIZACIÓN .....</b>	<b>- 40 -</b>
3.1. Sprint 0: Diario de planificación.....	- 40 -
3.2. Sprint 1: Análisis.....	- 41 -
3.3. Sprint 2: Investigación.....	- 42 -
3.4. Sprint 3-4: Diseño.....	- 43 -
3.5. Sprint 5-6: Codificación.....	- 44 -
<b>4. ANÁLISIS, DISEÑO E INFRAESTRUCTURA .....</b>	<b>- 46 -</b>
4.1. Análisis .....	- 46 -
4.1.1. Requisitos iniciales .....	- 46 -
4.1.2. Actores del sistema .....	- 47 -
4.1.3. Diagramas de casos de uso .....	- 48 -
4.2. Diseño .....	- 58 -
4.2.1. Diagrama de clases .....	- 58 -
4.2.2. Diagrama de secuencia .....	- 62 -
4.2.3. Bases de datos – Diagrama entidad / relación .....	- 68 -
4.2.4. Enfoque API First vs Code First .....	- 69 -
4.3. Infraestructura .....	- 70 -
4.3.1. Esquema de red .....	- 70 -
4.3.2. Flujo de llamadas entre servidores durante las peticiones .....	- 70 -
4.3.3. Docker .....	- 71 -
4.3.3.1. Servicio auth-db (PostgreSQL para autenticación) .....	- 72 -
4.3.3.2. Servicio course-psql-db (PostgreSQL para gestión de cursos).....	- 72 -
4.3.3.3. Servicio redis (Redis Stack para caché y colas ligeras).....	- 73 -
4.3.3.4. Servicio rabbit (RabbitMQ con consola de gestión).....	- 74 -
4.3.3.5. Red y volúmenes compartidos.....	- 76 -
<b>5. IMPLEMENTACIÓN DEL PROYECTO .....</b>	<b>- 77 -</b>
5.1. Aplicación con Go para autenticación y gestión de usuarios .....	- 77 -
5.1.1. ¿Por qué usar Go?.....	- 77 -



5.1.2. ¿Por qué usar UUID como tipo de dato para el identificador de los usuarios?.....	- 78 -
5.1.3. ¿Cómo generar el contrato Swagger a partir de los controladores?.....	- 79 -
5.1.4. ¿Cómo generar el Json Web Token y validararlo? .....	- 80 -
5.1.5. ¿Cómo conectar con la base de datos con GORM? .....	- 82 -
5.1.6. ¿Cómo conectar con él gestor de eventos RabbitMQ? .....	- 83 -
5.1.7. Arquitectura de 3 capas .....	- 84 -
5.1.8. Modelos de datos de dominio .....	- 86 -
5.1.9. Exposición de rutas y endpoints .....	- 87 -
5.1.10. Middleware previo a los controladores.....	- 89 -
5.1.11. Mensaje de manejo de errores.....	- 91 -
5.1.12. Funcionamiento de la aplicación .....	- 92 -
5.1.12.1. Registrar un usuario.....	- 93 -
5.1.12.2. Iniciar sesión .....	- 95 -
5.1.12.3. Cerrar sesión .....	- 97 -
5.1.12.4. Validar la sesión de un usuario .....	- 98 -
5.1.12.5. Ver perfil de un usuario .....	- 100 -
5.1.12.6. Eliminar un usuario .....	- 101 -
5.2. Aplicación con Java para la gestión de cursos .....	- 104 -
5.2.1. ¿Por qué usar Java con Spring Boot? .....	- 104 -
5.2.2. Arquitectura Hexagonal .....	- 105 -
5.2.2.1. ¿Por qué usar la arquitectura hexagonal en este proyecto?.....	- 106 -
5.2.2.2. ¿Por qué crear un proyecto multimódulo con Maven?.....	- 106 -
5.2.2.3. Funcionamiento .....	- 107 -
5.2.2.4. Estructura de carpetas/módulos .....	- 108 -
5.2.3. ¿Cómo generar el contrato Swagger a partir de los controladores?.....	- 108 -
5.2.4. ¿Cómo extraer la configuración de propiedades a las variables de entorno? .....	- 110 -
5.2.5. ¿Cómo conectar con la base de datos PostgreSQL y qué ORM usar? .....	- 111 -
5.2.6. ¿Cómo conectar con la base de datos Redis y qué ORM usar? .....	- 111 -
5.2.7. ¿Cómo conectar con el gestor de eventos RabbitMQ? .....	- 112 -
5.2.8. Control de versiones de la base de datos con FlyWay.....	- 113 -
5.2.9. Módulo de iniciación de datos (Patrón de diseño Data Seeding).....	- 114 -
5.2.10. Patrón saga .....	- 117 -
5.2.10.1. Publicadores de eventos.....	- 117 -



5.2.10.2. Eventos de dominio .....	- 118 -
5.2.10.3. Listeners de eventos y compensaciones.....	- 119 -
5.2.11. Patrón repository .....	- 120 -
5.2.12. Patrón assembler .....	- 121 -
5.2.13. Consumidor de eventos para RabbitMQ .....	- 122 -
5.2.14. Control de errores en RabbitMQ .....	- 124 -
5.2.15. Control de errores en REST .....	- 125 -
5.2.16. Configuración de roles y seguridad en REST .....	- 127 -
5.2.17. Funcionamiento de la aplicación .....	- 131 -
5.2.17.1. Ver todos los cursos .....	- 132 -
5.2.17.2. Ver los detalles de un curso .....	- 133 -
5.2.17.3. Insertar un curso .....	- 134 -
5.2.17.4. Actualizar un curso.....	- 135 -
5.2.17.5. Eliminar un curso .....	- 137 -
5.2.17.6. Inscribir un usuario en un curso.....	- 138 -
5.2.17.7. Completar una lección .....	- 140 -
5.2.17.8. ¿Qué sucede con el evento cuando un usuario se elimina en la app de auth?...-	141 -
<b>6. MANUAL DE USUARIO .....</b>	<b>- 142 -</b>
6.1. Requisitos.....	- 142 -
6.2. Postman .....	- 142 -
6.3. Pruebas .....	- 143 -
<b>7. PROPUESTAS DE MEJORA .....</b>	<b>- 152 -</b>
<b>8. VALORACIÓN PERSONAL .....</b>	<b>- 154 -</b>
<b>9. PUNTOS A DESTACAR .....</b>	<b>- 155 -</b>
<b>10. CONCLUSIÓN.....</b>	<b>- 157 -</b>
<b>11. BIBLIOGRAFÍA .....</b>	<b>- 158 -</b>

## 1. INTRODUCCIÓN

En la actualidad, el entorno digital evoluciona de forma acelerada, exigiendo que las soluciones tecnológicas no solo respondan con agilidad a las necesidades del usuario, sino que también sean seguras, escalables y fáciles de mantener. Esta dinámica impulsa el desarrollo de nuevas arquitecturas, patrones de diseño y metodologías que permiten construir software robusto y flexible, capaz de adaptarse a contextos cambiantes y con altos niveles de exigencia técnica.

Dentro de este contexto surge GoLearnix, una plataforma educativa que busca combinar la innovación pedagógica con las buenas prácticas del desarrollo moderno. Este trabajo se centra en el estudio, diseño e implementación de dicha plataforma, cuyo objetivo principal es servir como caso práctico para explorar arquitecturas avanzadas, modelos de diseño y separación de responsabilidades en el desarrollo de software.

GoLearnix se estructura en dos módulos principales: el sistema de autenticación, desarrollado en el lenguaje Go, y la gestión de cursos, implementada en Java. Ambos módulos están desacoplados y se comunican mediante un sistema de eventos, lo que permite una estructura más escalable y mantenible. En el lado de Java, se aplica el patrón CQRS (Command Query Responsibility Segregation), con el fin de separar claramente las operaciones de lectura y escritura. Además, se emplean dos bases de datos complementarias: PostgreSQL como base relacional y Redis como sistema NoSQL para mejorar el rendimiento en consultas rápidas.

La elección de estas tecnologías y patrones no es arbitraria. A lo largo de este trabajo se justifican cada una de ellas, analizando sus ventajas, limitaciones y adecuación al contexto de una aplicación educativa distribuida. Asimismo, se exploran distintas arquitecturas de software, como hexagonal, de microservicios y de sacrificio, evaluando su impacto en el diseño y mantenimiento del sistema.

Este documento se estructura en varias secciones para facilitar la comprensión de los objetivos planteados y las decisiones tomadas durante el desarrollo. En primer lugar, se introducen los conceptos fundamentales y el contexto tecnológico del proyecto. Posteriormente, se describe el diseño de la solución, sus componentes principales y la forma en que se integran. Se continúa con la justificación técnica de las tecnologías utilizadas y se detalla el proceso de implementación. Finalmente, se presentan las conclusiones extraídas del desarrollo del proyecto, así como posibles líneas de mejora y reflexión para trabajos futuros.



## 1.1. Entorno y Contextualización

El proyecto que se abordará, aunque podría ser aplicable a distintas instituciones educativas y plataformas de formación en línea, ha sido concebido específicamente como una solución educativa moderna bajo el nombre de GoLearnix. Esta plataforma se ha desarrollado en un entorno académico simulado, con el objetivo de servir tanto como sistema funcional de gestión de cursos como marco práctico para explorar buenas prácticas en el desarrollo de software web distribuido.

GoLearnix nace en un contexto donde la digitalización de la educación demanda soluciones escalables, seguras y mantenibles. A diferencia de otras propuestas centradas únicamente en aspectos pedagógicos o de experiencia de usuario, este proyecto pone el foco en el diseño de una arquitectura robusta, aplicando principios de separación de responsabilidades, uso de eventos, y patrones de diseño avanzados.

La implementación se ha llevado a cabo sobre una infraestructura compuesta por dos servicios principales: un módulo de autenticación, desarrollado en Go, y un módulo de gestión de cursos, desarrollado en Java. Como ya se ha comentado, ambos servicios están desacoplados y se comunican mediante un sistema basado en eventos, favoreciendo la escalabilidad y el bajo acoplamiento entre componentes. En el módulo de Java se aplica el patrón CQRS (Command Query Responsibility Segregation), permitiendo separar las operaciones de lectura y escritura para mejorar el rendimiento y la organización del código. A nivel de persistencia, se emplean dos bases de datos: una relacional con PostgreSQL para el almacenamiento estructurado de información, y una NoSQL, concretamente Redis, para la gestión eficiente de consultas rápidas y almacenamiento temporal.

Este entorno, por tanto, se presta a la experimentación y análisis de distintas arquitecturas, entre las que se destacan la de microservicios, la hexagonal y la de sacrificio. El propósito es evaluar sus fortalezas y debilidades, así como su aplicabilidad en proyectos reales que requieran alta disponibilidad, mantenimiento simplificado y facilidad de evolución.

El proyecto no se centra en soluciones de nivel hardware ni en aspectos puramente de infraestructura física, ni visuales, sino que su objetivo es plantear y validar alternativas basadas en software, estudiando a fondo los componentes necesarios para garantizar la fiabilidad, modularidad y eficiencia del sistema.



## 1.2. Objetivos

El presente trabajo nace de la necesidad de aplicar y demostrar buenas prácticas en el desarrollo de software moderno, especialmente en el contexto de aplicaciones educativas, donde la escalabilidad, modularidad y mantenibilidad del sistema son aspectos fundamentales. La creciente demanda de plataformas formativas accesibles y robustas exige soluciones técnicas que garanticen un funcionamiento fiable y adaptable a distintos escenarios.

Por tanto, el propósito principal consiste en diseñar e implementar un producto mínimo viable de una plataforma educativa que refleje principios avanzados de arquitectura de software, haciendo hincapié en la separación de responsabilidades entre los diferentes módulos que la componen, como la autenticación y la gestión de cursos, desarrollados respectivamente en Go y Java.

Uno de los objetivos clave es integrar un sistema de comunicación basado en eventos entre servicios, que permita un acoplamiento bajo, mayor escalabilidad y una mejor gestión de flujos asincrónicos. En paralelo, se busca aplicar el patrón CQRS en el módulo de gestión de cursos para optimizar la separación entre operaciones de lectura y escritura, y así mejorar el rendimiento y la organización del sistema.

Asimismo, se pretende evaluar distintas arquitecturas de software, como microservicios, hexagonal y de sacrificio, identificando sus ventajas y desventajas en un entorno realista. Otro objetivo importante es la integración de múltiples tecnologías de persistencia, como PostgreSQL (base de datos relacional) y Redis (NoSQL), y la justificación técnica de su uso combinado dentro del sistema.

En definitiva, este proyecto tiene como finalidad ofrecer una solución educativa funcional y moderna, que sirva tanto como herramienta de formación como ejemplo práctico de cómo diseñar software distribuido aplicando principios sólidos de ingeniería. Se buscará garantizar que los componentes del sistema sean modulares, escalables y fácilmente mantenibles, sentando las bases para futuras ampliaciones o integraciones en otros entornos educativos o profesionales.



## 2. MARCO CONCEPTUAL

### Java / Spring Boot

La ingeniería de software, como disciplina fundamental de las ciencias de la computación, se centra en la aplicación de principios y metodologías para el desarrollo y mantenimiento de sistemas de software confiables y de calidad. En este contexto, el lenguaje de programación Java y el framework Spring Boot han emergido como herramientas esenciales para la creación de aplicaciones modernas, especialmente en arquitecturas basadas en microservicios.

**Java** es un lenguaje de programación orientado a objetos, ampliamente utilizado en el desarrollo de aplicaciones empresariales debido a su portabilidad, robustez y amplia comunidad de desarrolladores. Su sintaxis clara y su modelo de ejecución basado en la Máquina Virtual de Java (JVM) permiten la creación de aplicaciones escalables y mantenibles.

En este ámbito, Java ha sido adoptado como lenguaje principal en numerosos proyectos debido a su capacidad para integrarse con diversas tecnologías y frameworks, facilitando así el cumplimiento de los requisitos de calidad y eficiencia en el desarrollo de software.

El Spring Framework es un conjunto de herramientas y librerías para el desarrollo de aplicaciones Java, que proporciona soluciones para la inyección de dependencias, programación orientada a aspectos, acceso a datos y más. Sin embargo, su configuración inicial puede ser compleja y propensa a errores.

Para simplificar este proceso, se desarrolló **Spring Boot**, una extensión del Spring Framework que permite crear aplicaciones independientes y listas para producción con una configuración mínima. Spring Boot sigue el principio de "convención sobre configuración", proporcionando configuraciones predeterminadas que permiten a los desarrolladores centrarse en la lógica de negocio.



GoLang / Fiber

Go, comúnmente conocido como **GoLang**, es un lenguaje de programación desarrollado por **Google** en 2007 y lanzado públicamente en 2009. Diseñado para abordar las deficiencias de otros lenguajes en cuanto a **simplicidad, eficiencia y concurrencia**, Go se ha consolidado como una opción preferente en el desarrollo de sistemas distribuidos y aplicaciones web de alto rendimiento.

Una de las características distintivas de Go es su modelo de concurrencia basado en goroutines y canales, que permite la ejecución simultánea de múltiples procesos de manera eficiente y segura. Este enfoque facilita la construcción de aplicaciones escalables y concurrentes, esenciales en la arquitectura moderna de microservicios.

**Fiber** es un framework web para Go que se inspira en la simplicidad y eficiencia de Express.js de Node.js. Construido sobre el motor HTTP de alto rendimiento Fasthttp, Fiber ofrece una estructura ligera y rápida para el desarrollo de aplicaciones web. Su diseño modular y su API intuitiva permiten a los desarrolladores crear aplicaciones robustas con una curva de aprendizaje reducida.

Una de las implementaciones destacadas de Fiber es su adherencia a los principios de la "Clean Architecture", propuesta por Robert C. Martin, la cual se abordará más adelante.

Además, Fiber incorpora un sistema de middleware que permite la ejecución de funciones específicas antes o después del procesamiento de una solicitud. Esto incluye funcionalidades como autenticación, manejo de errores y registro de actividades, contribuyendo a una arquitectura limpia y modular.

Fiber se diferencia de otros frameworks por su enfoque en la velocidad y la simplicidad, siendo una opción preferida para aplicaciones que demandan alto rendimiento y una estructura ligera.



### Diseño guiado por el dominio

El Diseño Guiado por el Dominio (**Domain-Driven Design, DDD**) es una metodología que busca alinear el diseño del sistema con el conocimiento profundo del dominio del negocio. Propuesto por Eric Evans en su obra seminal *Domain-Driven Design: Tackling Complexity in the Heart of Software* (2004), DDD se centra en abordar la complejidad inherente de los sistemas mediante una comprensión detallada del dominio y una colaboración estrecha entre expertos del negocio y desarrolladores.

Se basa en la premisa de que el diseño de software debe reflejar con precisión el dominio del negocio al que sirve. Para lograr esto, se promueve la creación de un "*Lenguaje Ubícuo*" (Ubiquitous Language).

Además, DDD introduce el concepto de "*Contexto Delimitado*" (Bounded Context), que define límites explícitos dentro de los cuales un modelo particular es aplicable. Esto permite manejar la complejidad dividiendo el sistema en subdominios coherentes, facilitando la evolución independiente y evitando ambigüedades en la interpretación de los modelos.

DDD distingue entre dos niveles de diseño:

- **Diseño Estratégico:** Se enfoca en la arquitectura general del sistema, identificando subdominios y estableciendo los Contextos Delimitados. Este nivel guía las decisiones sobre cómo dividir el sistema y cómo interactúan las diferentes partes entre sí.
- **Diseño Táctico:** Proporciona patrones y prácticas para implementar los modelos dentro de cada Contexto Delimitado. Incluye conceptos como Entidades, Objetos de Valor, Agregados, Repositorios y Servicios, que ayudan a estructurar el código de manera coherente con el dominio.

La adopción de DDD ha demostrado ser particularmente efectiva en arquitecturas de microservicios, donde cada servicio puede alinearse con un Contexto Delimitado específico. Esto facilita la escalabilidad, el mantenimiento y la evolución del sistema, al tiempo que permite una mayor autonomía de los equipos de desarrollo.

## Arquitectura limpia

La **Arquitectura Limpia**, propuesta por Robert C. Martin (también conocido como "Uncle Bob"), representa un paradigma que busca lograr una alta mantenibilidad, escalabilidad y testabilidad. Este enfoque **se centra en la separación de responsabilidades y la independencia de los detalles de implementación**, permitiendo que la lógica de negocio permanezca aislada de factores externos como frameworks, bases de datos o interfaces de usuario.

Se basa en varios principios fundamentales que garantizan la modularidad y la independencia de los componentes del software:

- **Independencia de Frameworks:** La aplicación no debe depender de frameworks específicos; estos deben ser herramientas, no dictar la arquitectura.
- **Independencia de la Interfaz de Usuario:** La UI puede cambiar sin afectar la lógica de negocio, permitiendo diferentes interfaces (web, móvil, CLI) sin alterar el núcleo.
- **Independencia de la Base de Datos:** Es posible cambiar la base de datos sin modificar la lógica de negocio, ya que esta no debe depender de detalles de almacenamiento.
- **Independencia de Detalles Externos:** La lógica de negocio es agnóstica a frameworks, bibliotecas o herramientas externas.

Estos principios se alinean con los principios SOLID, los cuales se verán más adelante.

Se organiza en capas concéntricas, donde cada capa tiene reglas claras sobre cómo interactuar con las demás:

- **Entidades (Core/Enterprise Business Rules):** Representan las reglas de negocio más generales y son independientes de cualquier tecnología o framework.
- **Casos de Uso (Application Business Rules):** Contienen la lógica específica de aplicación y orquestan el comportamiento del sistema según los requerimientos del negocio.
- **Adaptadores de Interfaz (Interface Adapters):** Transforman datos entre la capa de aplicación y los controladores, la UI, bases de datos o APIs externas.
- **Infraestructura y Frameworks (Frameworks & Drivers):** Contiene herramientas externas como bases de datos, frameworks web y bibliotecas de terceros, sin incluir lógica de negocio.

La regla de la independencia establece que las dependencias deben apuntar solo hacia adentro, hacia las políticas de alto nivel. Ningún componente de un círculo interno debe saber nada de un círculo más exterior, lo que incluye funciones, clases, variables o cualquier otra entidad de software nombrada.

### Arquitectura de tres capas

La **arquitectura de tres capas**, también conocida como arquitectura de tres niveles, organiza una aplicación en tres componentes lógicos y físicos:

- **Capa de Presentación:** Encargada de la interfaz de usuario, gestiona la interacción con el usuario final y presenta la información de manera comprensible.
- **Capa de Lógica de Negocio:** Contiene las reglas y procesos que definen el comportamiento del sistema, procesando las solicitudes de la capa de presentación y determinando cómo deben interactuar con los datos.
- **Capa de Datos:** Responsable del almacenamiento y recuperación de datos, interactúa con sistemas de gestión de bases de datos para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar).

Esta separación permite que cada capa se desarrolle, mantenga y escale de forma independiente, mejorando la modularidad y facilitando el trabajo en equipo.

El concepto de separación en capas se remonta a la arquitectura ANSI-SPARC propuesta en 1975, que establecía una división en niveles externo, conceptual e interno para sistemas de bases de datos. Esta idea evolucionó y se adaptó al desarrollo de software en general, dando lugar a la arquitectura de tres capas como se conoce hoy.

En la década de 1990, con el auge de las aplicaciones cliente-servidor, la arquitectura de tres capas se consolidó como un estándar para el desarrollo de sistemas distribuidos, permitiendo una mejor gestión de la complejidad y facilitando la adopción de nuevas tecnologías en cada capa.



En el contexto actual, donde la agilidad y la capacidad de adaptación son cruciales, la arquitectura de tres capas sigue siendo una herramienta valiosa para ingenieros de software y arquitectos de sistemas.

### Arquitectura hexagonal

La **arquitectura hexagonal**, también conocida como arquitectura de puertos y adaptadores, representa un paradigma de diseño de software que **busca desacoplar la lógica de negocio del sistema de sus dependencias externas**. Propuesta por Alistair Cockburn en 2005, esta arquitectura surge como respuesta a las limitaciones observadas en los modelos tradicionales, como la arquitectura en capas, donde las dependencias entre componentes pueden generar rigidez y dificultades en la evolución del software.

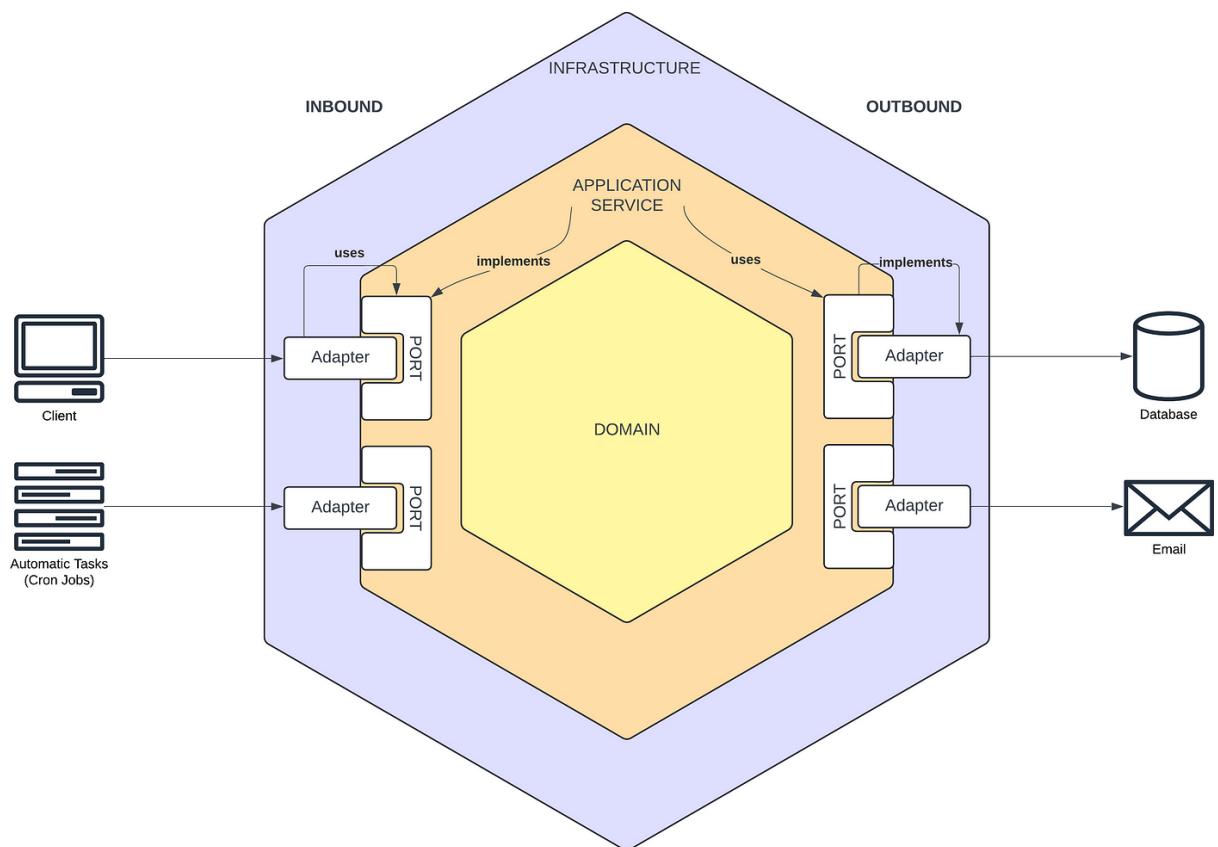
El núcleo de la arquitectura hexagonal es el dominio de la aplicación, que encapsula la lógica de negocio y las reglas fundamentales del sistema. Este núcleo es completamente independiente de cualquier tecnología o infraestructura externa. Para interactuar con el mundo exterior, se utilizan puertos y adaptadores:

- **Puertos:** Son interfaces que definen cómo interactuar con el núcleo de la aplicación. Se dividen en:
  - **Puertos de entrada:** Permiten que el exterior interactúe con la aplicación (por ejemplo, a través de controladores de una API o eventos).
  - **Puertos de salida:** Permiten que la aplicación interactúe con servicios externos (como bases de datos o APIs de terceros).
- **Adaptadores:** Son implementaciones concretas de los puertos. Se encargan de traducir las interacciones entre el núcleo y el mundo exterior. Se clasifican en:
  - **Adaptadores de entrada:** Transforman las solicitudes externas en comandos comprensibles para el núcleo.
  - **Adaptadores de salida:** Implementan las interacciones con servicios externos.

La representación gráfica de esta arquitectura utiliza un hexágono para simbolizar el núcleo, con cada lado representando un puerto. Esta forma no implica un número fijo de puertos, sino que enfatiza la modularidad y la capacidad de interactuar con múltiples interfaces externas.



La arquitectura hexagonal es especialmente útil en sistemas donde la lógica de negocio es compleja y se requiere una alta flexibilidad para adaptarse a cambios en las tecnologías externas. Su implementación es común en aplicaciones empresariales, sistemas con múltiples interfaces de usuario o integraciones con diversos servicios externos. Además, su compatibilidad con principios de diseño como Domain-Driven Design (DDD) la convierte en una opción robusta para el desarrollo de software de calidad.



### Arquitectura monolítica

La **arquitectura monolítica** representa uno de los paradigmas más tradicionales en el desarrollo de sistemas de software. **Caracterizada por la integración de todos los componentes funcionales en una única unidad indivisible**, esta arquitectura ha sido ampliamente adoptada en diversas aplicaciones, especialmente en contextos donde la simplicidad y la cohesión son prioritarias. A pesar de la creciente popularidad de arquitecturas más modulares y distribuidas, como los microservicios, la arquitectura monolítica sigue siendo relevante, ofreciendo ventajas significativas en términos de desarrollo y despliegue.

Una aplicación monolítica se construye como una única unidad ejecutable que encapsula todas las funcionalidades necesarias: interfaz de usuario, lógica de negocio, acceso a datos y otros componentes esenciales. Esta estructura implica que todos los módulos están estrechamente acoplados y comparten un espacio de memoria común, facilitando la comunicación interna mediante llamadas directas a funciones o procedimientos.

No obstante, con el auge de las arquitecturas basadas en microservicios, muchas organizaciones han optado por migrar desde estructuras monolíticas hacia sistemas más modulares y escalables. Sin embargo, esta transición no siempre es sencilla ni necesaria. En algunos casos, se ha propuesto la adopción de arquitecturas monolíticas modulares, que combinan la simplicidad del monolito con ciertos beneficios de los microservicios, como la separación de responsabilidades y la escalabilidad parcial.

## Arquitectura de microservicios

La **arquitectura de microservicios** representa una evolución significativa, ofreciendo una alternativa moderna a las arquitecturas monolíticas tradicionales. Este enfoque **se centra en la descomposición de aplicaciones complejas en servicios pequeños, autónomos y especializados, cada uno de los cuales se encarga de una funcionalidad específica del negocio**. Estos servicios se comunican entre sí mediante interfaces bien definidas, generalmente a través de APIs ligeras o eventos, permitiendo una mayor flexibilidad, escalabilidad y resiliencia en los sistemas.

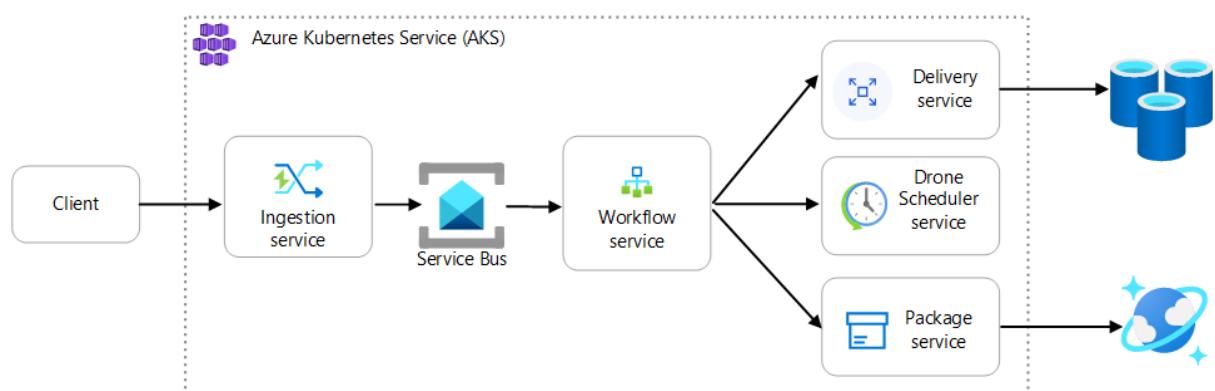
La arquitectura de microservicios se basa en principios fundamentales como el Principio de Responsabilidad Única (SRP), que establece que cada servicio debe tener una única responsabilidad o función dentro del sistema. Este principio promueve una alta cohesión dentro de los servicios y un bajo acoplamiento entre ellos, facilitando su desarrollo, mantenimiento y escalado independiente.

Además, la arquitectura de microservicios está estrechamente relacionada con el Diseño Guiado por el Dominio (DDD). El DDD enfatiza la importancia de modelar el software en torno a los conceptos y procesos del dominio del negocio, lo que se traduce en servicios que reflejan contextos delimitados y alineados con las capacidades empresariales.



A diferencia de las arquitecturas monolíticas, donde todos los componentes de una aplicación están integrados en un único bloque de código, la arquitectura de microservicios promueve la modularidad y la separación de preocupaciones. Esto facilita la comprensión del sistema, mejora la mantenibilidad y permite una evolución más ágil del software. Sin embargo, también introduce desafíos adicionales, como la complejidad en la gestión de múltiples servicios, la necesidad de una infraestructura adecuada para el despliegue y la orquestación, y la implementación de mecanismos de comunicación y seguridad entre servicios.

Al adoptar este enfoque, las organizaciones pueden mejorar su capacidad para responder a los cambios del mercado, innovar rápidamente y ofrecer soluciones de software más robustas y adaptables.



### Arquitectura guiada por eventos

**La arquitectura guiada por eventos se basa en la producción, detección y consumo de eventos para facilitar la comunicación entre componentes de software.** Un evento se define como un cambio significativo en el estado de un sistema. En este contexto, los eventos actúan como señales que desencadenan acciones específicas dentro del sistema.

Los componentes principales en una arquitectura guiada por eventos incluyen:

- **Productores:** generan eventos en respuesta a cambios de estado o acciones específicas.
- **Consumidores:** procesan los eventos recibidos y ejecutan acciones en consecuencia.
- **Canales:** medios a través de los cuales los eventos se transmiten desde los productores a los consumidores, como colas de mensajes o flujos de eventos.

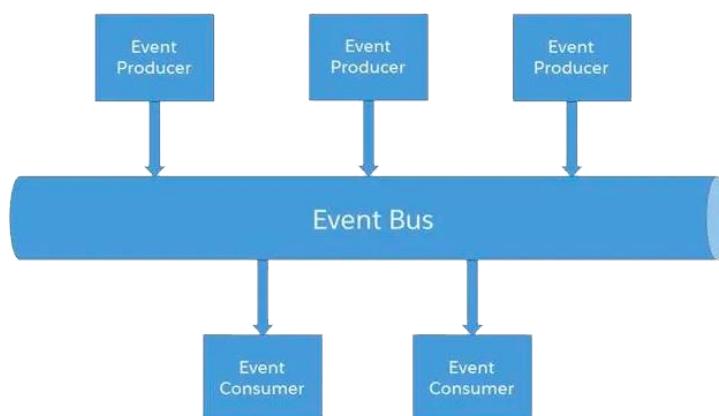
- **Brokers:** intermediarios que gestionan la distribución y entrega de eventos, garantizando la fiabilidad y eficiencia en la comunicación.

La arquitectura guiada por eventos se caracteriza por varios principios clave que la distinguen de otros paradigmas arquitectónicos:

- **Desacoplamiento:** los componentes interactúan mediante eventos sin depender directamente unos de otros, lo que permite una mayor flexibilidad y escalabilidad en el sistema.
- **Asincronía:** la comunicación basada en eventos es inherentemente asincrónica, lo que permite a los componentes operar de manera independiente y mejorar la capacidad de respuesta del sistema.
- **Procesamiento en tiempo real:** los sistemas pueden reaccionar a eventos a medida que ocurren, facilitando la toma de decisiones en tiempo real y la adaptación dinámica a cambios en el entorno.

Además, se han identificado varios patrones arquitectónicos asociados con la EDA (Arquitectura Dirigida por eventos):

- **Event Sourcing:** en lugar de almacenar el estado actual de una entidad, se almacenan todos los eventos que han llevado a ese estado, permitiendo reconstruir el historial completo y facilitar auditorías.
- **CQRS (Command Query Responsibility Segregation):** separación de las operaciones de lectura y escritura en modelos distintos, optimizando el rendimiento y la escalabilidad del sistema. Más adelante se verá con más detalles.



## Arquitectura de sacrificio

La "**arquitectura de sacrificio**" es un enfoque estratégico que **implica diseñar sistemas con la aceptación anticipada de que, en un futuro, partes o la totalidad del sistema serán reemplazadas**.

Este concepto fue introducido por Martin Fowler, quien destaca la importancia de construir sistemas que puedan ser descartados y reemplazados cuando las necesidades cambien o evolucionen.

Este enfoque reconoce que, debido a la rápida evolución tecnológica y a las cambiantes demandas del mercado, es más eficiente y sostenible diseñar sistemas que puedan ser fácilmente modificados o reemplazados. Así, se evita la rigidez de sistemas monolíticos que dificultan la adaptación y escalabilidad.

Un ejemplo notable de la aplicación de la arquitectura de sacrificio es el caso de eBay, que ha reconstruido su sistema varias veces para adaptarse a nuevas tecnologías y demandas. Twitter también ha rediseñado su API pública para mejorar la escalabilidad y consistencia, aprendiendo de experiencias pasadas.

La modularidad es un principio clave en este enfoque. Diseñar sistemas modulares permite reemplazar o actualizar componentes individuales sin afectar el sistema completo, facilitando la evolución y mantenimiento del software.

En otras palabras, la arquitectura de sacrificio es una estrategia que promueve la flexibilidad y adaptabilidad en el diseño de sistemas, aceptando el cambio como una constante y preparando el software para evolucionar con las necesidades futuras.

## Patrón de diseño repository

Los patrones de diseño constituyen soluciones reutilizables a problemas comunes en el desarrollo de sistemas. Entre ellos, el **patrón Repository** se destaca por su capacidad para abstraer y encapsular el acceso a los datos, promoviendo una arquitectura más modular, mantenable y testable.



**Actúa como una interfaz entre la capa de dominio y la capa de acceso a datos, proporcionando una colección de métodos para acceder y manipular objetos del dominio sin exponer los detalles de la infraestructura subyacente.** Según Martin Fowler, este patrón "*mediatiza entre el dominio y las capas de mapeo de datos utilizando una interfaz similar a una colección para acceder a objetos del dominio*" (Fowler, 2002). La principal motivación es desacoplar la lógica de negocio de las preocupaciones relacionadas con la persistencia, facilitando así la evolución y el mantenimiento del sistema.

Aunque fue popularizado por Martin Fowler en su obra "Patterns of Enterprise Application Architecture" (2002), sus raíces se remontan a trabajos anteriores sobre arquitectura de software. Por ejemplo, David Garlan y Mary Shaw ya discutían conceptos similares en su artículo "An Introduction to Software Architecture" (1994), donde exploraban la necesidad de separar las preocupaciones del dominio y la infraestructura. Posteriormente, Eric Evans consolidó el uso del patrón Repository en el contexto del diseño dirigido por el dominio (DDD), enfatizando su papel en la gestión de agregados y la coherencia del modelo de dominio (Evans, 2003).

En la práctica, un repositorio define una interfaz con métodos como create, delete, y find, que permiten manipular objetos del dominio sin exponer los detalles de la persistencia.

Es importante destacar que un repositorio generalmente gestiona un único tipo de agregado, actuando como una colección de objetos del dominio. Esto facilita la coherencia y la integridad de los datos, ya que todas las operaciones relacionadas con un agregado específico se centralizan en un único punto de acceso.

Además de todo esto, es útil en aplicaciones que requieren soportar múltiples tecnologías de persistencia, ya que permite abstraer las diferencias entre ellas y proporcionar una interfaz uniforme para el acceso a los datos.

### Patrón saga

La evolución hacia arquitecturas de microservicios ha transformado la manera en que se diseñan y gestionan las aplicaciones distribuidas. Este paradigma promueve la descomposición de sistemas monolíticos en servicios autónomos que interactúan entre sí, lo que introduce desafíos

significativos en la gestión de transacciones distribuidas y la consistencia de datos. En este escenario, el patrón Saga emerge como una solución arquitectónica clave para abordar estas complejidades, permitiendo coordinar transacciones distribuidas de manera eficiente y resiliente.

El **concepto de Saga** fue introducido por Hector García-Molina y Kenneth Salem en 1987, quienes propusieron una estrategia para manejar transacciones de larga duración en sistemas distribuidos mediante **la descomposición de una transacción global en una secuencia de transacciones locales, cada una con su correspondiente acción de compensación en caso de fallo**. Este enfoque permitía mantener la consistencia de los datos sin necesidad de bloquear recursos durante períodos prolongados, como ocurre en los protocolos tradicionales de commit en dos fases.

Con la adopción masiva de arquitecturas de microservicios, el patrón Saga ha resurgido como una práctica esencial para gestionar la consistencia eventual en sistemas donde las transacciones abarcan múltiples servicios independientes.

Se basa en la idea de dividir una transacción global en una serie de transacciones locales que se ejecutan de forma secuencial. Cada transacción local realiza una operación específica y, en caso de éxito, desencadena la siguiente transacción en la secuencia. Si alguna transacción falla, se ejecutan acciones de compensación para revertir los efectos de las transacciones anteriores, asegurando así la consistencia del sistema.

Existen dos enfoques principales para implementar el patrón Saga:

- **Coreografía:** En este modelo descentralizado, cada servicio participante en la saga escucha eventos y, al recibir uno relevante, ejecuta su transacción local y emite un nuevo evento para el siguiente servicio. Este enfoque promueve un acoplamiento débil entre servicios y facilita la escalabilidad, pero puede complicar el seguimiento del flujo de transacciones y la gestión de errores.
- **Orquestación:** Este modelo centralizado introduce un componente coordinador que gestiona la secuencia de transacciones, enviando comandos a los servicios participantes y manejando las respuestas. Aunque proporciona un control más explícito sobre el flujo de la saga y simplifica la gestión de errores, introduce un punto único de fallo y puede aumentar la complejidad del sistema.

### Patrón assembler

El patrón de diseño conocido como "**Assembler**" desempeña un papel crucial en contextos donde **se requiere la transformación y adaptación de objetos entre diferentes capas o dominios de una aplicación**. Este patrón se enmarca en los patrones estructurales, cuya finalidad es facilitar la composición de clases y objetos para formar estructuras más complejas, promoviendo así la reutilización y la flexibilidad en el diseño del software.

Se utiliza para convertir objetos de un tipo a otro, generalmente entre objetos de dominio y objetos de transferencia de datos (DTOs, por sus siglas en inglés). Esta conversión es esencial en aplicaciones multicapa, donde cada capa tiene responsabilidades distintas y, por ende, representaciones de datos diferentes. El Assembler actúa como un traductor que permite que estas capas se comuniquen sin acoplarse directamente, lo que favorece la mantenibilidad y escalabilidad del sistema.

La implementación típica implica la creación de una clase específica que contiene métodos para convertir objetos de un tipo a otro. Estos métodos toman un objeto de entrada y devuelven un nuevo objeto del tipo deseado, mapeando los atributos correspondientes según sea necesario.

### Patrón CQRS

El patrón CQRS se basa en el principio de Separación de Comandos y Consultas (**Command-Query Separation, CQS**), introducido por Bertrand Meyer en el contexto de la programación orientada a objetos. Meyer argumenta que un método debe ser o un comando, que modifica el estado del sistema, o una consulta, que devuelve datos sin alterar el estado. CQRS extiende este principio a nivel arquitectónico, **separando las responsabilidades de lectura y escritura en diferentes modelos y, en ocasiones, en diferentes servicios**.

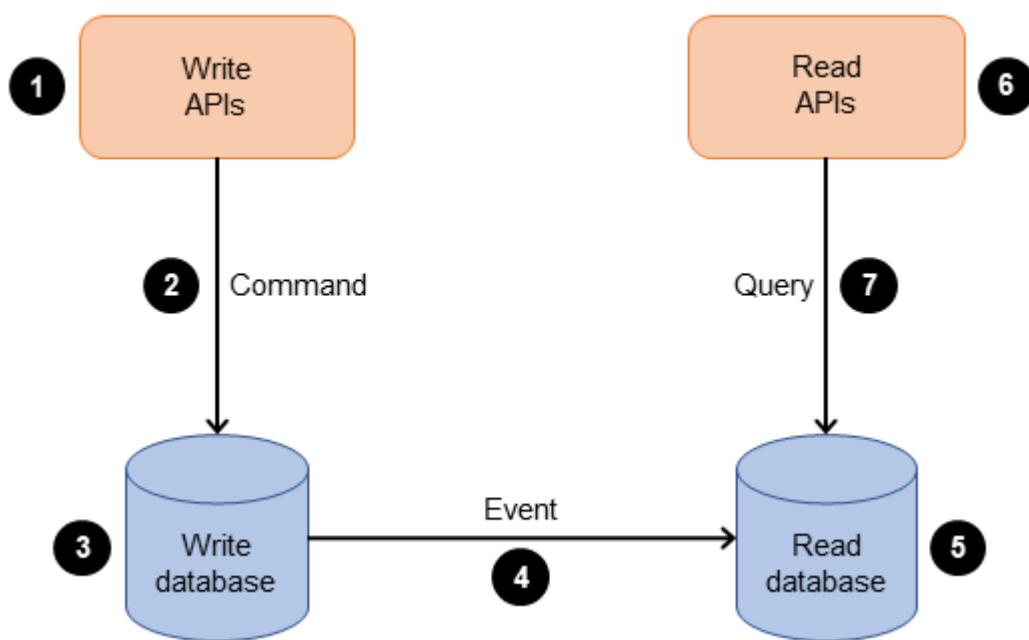
Aunque Greg Young es ampliamente reconocido por formalizar y popularizar el patrón CQRS en 2010, se reconoce que Udi Dahan ya había explorado conceptos similares en 2008, en el contexto de la Arquitectura Orientada a Servicios (SOA) y el Diseño Guiado por el Dominio (DDD).



El patrón CQRS se caracteriza por la separación de las operaciones de lectura y escritura en componentes distintos:

- **Modelo de Comandos (Write Model):** Encargado de procesar las operaciones que modifican el estado del sistema. Este modelo implementa la lógica de negocio y las reglas de validación necesarias para garantizar la integridad de los datos.
- **Modelo de Consultas (Read Model):** Responsable de manejar las operaciones de lectura. Este modelo está optimizado para consultas eficientes y puede utilizar estructuras de datos denormalizadas o bases de datos especializadas para mejorar el rendimiento.

La separación de estos modelos permite que cada uno evolucione y escale de manera independiente, adaptándose a las necesidades específicas de lectura o escritura del sistema.



### Principio de modularización de configuración

**La modularización implica dividir un sistema en componentes independientes denominados módulos, cada uno encargado de una funcionalidad específica.** Esta estrategia se basa en principios como la alta cohesión y el bajo acoplamiento, que buscan que los módulos sean internamente coherentes y dependan mínimamente entre sí. Autores como Bertrand Meyer y

Barbara Liskov han destacado la importancia de la modularidad para la comprensión y mantenimiento de sistemas complejos.

Además, la modularización facilita la aplicación de principios de diseño como SOLID, que promueven la creación de sistemas más robustos y flexibles. Por ejemplo, el principio de única responsabilidad sugiere que cada módulo debe tener una razón única para cambiar, mientras que el principio de inversión de dependencias aboga por depender de abstracciones y no de detalles concretos, el cual se verá con más detalle en las siguientes secciones.

### Principios SOLID

Los **principios SOLID** constituyen un conjunto de directrices fundamentales para el diseño y desarrollo de sistemas orientados a objetos. Estos principios fueron formalizados por Robert C. Martin, conocido como "Uncle Bob", a principios de la década de 2000, y el acrónimo SOLID fue acuñado posteriormente por Michael Feathers. La aplicación de estos principios busca mejorar la mantenibilidad, escalabilidad y robustez del software, facilitando su adaptación a cambios y reduciendo la complejidad inherente al desarrollo de sistemas complejos.

- **Principio de Responsabilidad Única (Single Responsibility Principle - SRP):**

Este principio establece que **una clase debe tener una única razón para cambiar**, es decir, debe estar enfocada en una sola responsabilidad o función dentro del sistema. Al adherirse al SRP, se promueve una alta cohesión y un bajo acoplamiento, facilitando la comprensión, mantenimiento y prueba del código. Una clase que asume múltiples responsabilidades se vuelve más propensa a errores y más difícil de modificar sin afectar otras partes del sistema

- **Principio Abierto/Cerrado (Open/Closed Principle - OCP)**

Formulado por Bertrand Meyer en 1988, sostiene que **las entidades de software deben estar abiertas para su extensión, pero cerradas para su modificación**. Esto implica que el comportamiento de una clase puede ser extendido sin alterar su código fuente, utilizando



mecanismos como la herencia o la composición. La aplicación del OCP permite agregar nuevas funcionalidades sin introducir errores en el comportamiento existente, promoviendo la reutilización y la estabilidad del sistema.

- **Principio de Sustitución de Liskov (Liskov Substitution Principle - LSP)**

Introducido por Barbara Liskov en 1987, establece que **los objetos de una clase derivada deben poder sustituir a los objetos de su clase base sin alterar el correcto funcionamiento** del programa. El LSP garantiza que las subclases mantengan las expectativas establecidas por sus superclases, asegurando la coherencia y la integridad del sistema al utilizar la herencia.

- **Principio de Segregación de Interfaces (Interface Segregation Principle - ISP)**

El ISP propone que **los clientes no deben verse obligados a depender de interfaces que no utilizan**. En lugar de interfaces generales y extensas, se recomienda diseñar interfaces específicas y enfocadas en las necesidades particulares de cada cliente. Esto reduce el acoplamiento y mejora la flexibilidad del sistema, permitiendo que las clases implementen solo los métodos que realmente necesitan.

- **Principio de Inversión de Dependencias (Dependency Inversion Principle - DIP)**

Sugiere que **los módulos de alto nivel no deben depender de módulos de bajo nivel; ambos deben depender de abstracciones**. Además, las abstracciones no deben depender de detalles, sino que los detalles deben depender de las abstracciones. La implementación del DIP promueve un diseño desacoplado y flexible, facilitando la reutilización y el mantenimiento del código.

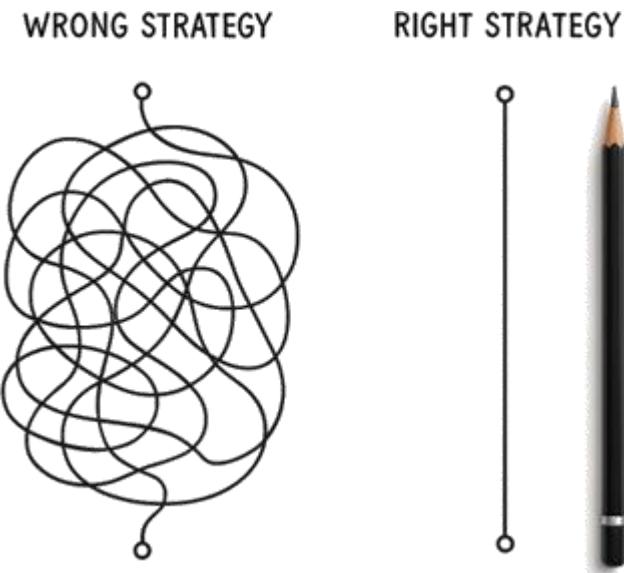
## Principios KISS, DRY y YAGNI

Principios como KISS (Keep It Simple, Stupid), DRY (Don't Repeat Yourself) y YAGNI (You Aren't Gonna Need It), los cuales promueven la simplicidad, la reducción de redundancias y la implementación de funcionalidades estrictamente necesarias.

- **Principio KISS: "Keep It Simple, Stupid"**



El principio KISS, acrónimo de "Keep It Simple, Stupid", **enfatiza la importancia de mantener la simplicidad en el diseño y desarrollo de sistemas**. Fue popularizado por Kelly Johnson, ingeniero jefe de Lockheed Skunk Works, quien abogaba por soluciones simples que pudieran ser comprendidas y mantenidas fácilmente por técnicos en condiciones adversas. En el contexto del desarrollo de software, KISS sugiere que las soluciones deben ser lo más simples posible, evitando complejidades innecesarias que puedan dificultar la comprensión y el mantenimiento del código.



- **Principio DRY: "Don't Repeat Yourself"**

El principio DRY, formulado por Andy Hunt y Dave Thomas en su libro *The Pragmatic Programmer*, establece que "*cada pieza de conocimiento debe tener una representación única, inequívoca y autoritativa dentro de un sistema*". **Busca eliminar la redundancia en el código, promoviendo la reutilización y la coherencia en la implementación de funcionalidades.**

La implementación del principio DRY puede lograrse mediante el uso de funciones reutilizables, clases bien definidas y estructuras de datos adecuadas que promuevan la modularidad y la cohesión del sistema.

- **Principio YAGNI: "You Aren't Gonna Need It"**

YAGNI, originado en la metodología de programación extrema (Extreme Programming), sostiene que "*no debes agregar funcionalidad a menos que sea necesaria*". Este enfoque **busca evitar la sobreingeniería y la incorporación de características que no aportan valor inmediato al sistema**.

En conjunto, estos principios apoyan prácticas de desarrollo ágil, donde la adaptabilidad, la eficiencia y la calidad del software son prioritarias. Su implementación requiere una comprensión profunda de los objetivos del proyecto, así como una comunicación efectiva dentro del equipo de desarrollo para asegurar que las decisiones de diseño y codificación estén alineadas con estos principios.

### Objeto de transferencia de datos

Un Objeto de Transferencia de Datos (**DTO**) es un patrón de diseño estructural utilizado para transferir datos entre subsistemas de una aplicación, especialmente en arquitecturas distribuidas. Según Martin Fowler, el propósito principal de un DTO es reducir el número de llamadas remotas al agrupar múltiples datos en un solo objeto, optimizando así la comunicación entre cliente y servidor.

Los DTOs son objetos simples que contienen solo datos y carecen de lógica de negocio. Su función principal es transportar datos entre procesos, minimizando la sobrecarga de red y mejorando el rendimiento en sistemas distribuidos.

### Modelos de datos como proyecciones

El modelado de datos constituye una fase esencial en el desarrollo de sistemas de información, permitiendo representar de manera estructurada los elementos y relaciones del dominio de aplicación. Dentro de este proceso, las proyecciones de modelos de datos emergen como una técnica fundamental para abstraer y manipular subconjuntos específicos de información, facilitando la eficiencia y claridad en la gestión de datos.

**Las proyecciones de modelos de datos se refieren a la operación mediante la cual se seleccionan atributos específicos de una entidad o conjunto de datos**, generando una vista parcial que responde a necesidades particulares del sistema o del usuario. Este concepto tiene sus raíces en el álgebra relacional, donde la proyección (*representada por el operador  $\pi$* ) permite obtener una

relación con un subconjunto de las columnas de una tabla, eliminando duplicados y enfocándose en la información relevante.

En el contexto de bases de datos relacionales, la proyección se implementa comúnmente mediante la instrucción SELECT en SQL, permitiendo recuperar columnas específicas de una tabla. Esta operación no solo optimiza el rendimiento al reducir la cantidad de datos procesados y transferidos, sino que también contribuye a la seguridad al limitar el acceso a información sensible.

El modelado de datos se estructura en tres niveles:

- **Modelo Conceptual:** Representa de manera abstracta las entidades y relaciones del dominio, sin considerar detalles técnicos.
- **Modelo Lógico:** Detalla las estructuras de datos y sus relaciones, adaptándolas a un sistema de gestión de bases de datos específico.
- **Modelo Físico:** Define la implementación concreta de los datos en el sistema, incluyendo aspectos como tipos de datos y almacenamiento.

Las proyecciones se aplican principalmente en los niveles lógico y físico, permitiendo adaptar y optimizar la representación de los datos según las necesidades del sistema y las restricciones tecnológicas.

### API First y Code First

El enfoque **API First**, también conocido como Design First o Contract First, **se centra en definir las APIs desde el inicio del proceso de desarrollo**. Esto implica crear una especificación detallada de la API antes de implementar cualquier código, estableciendo un contrato claro entre los diferentes componentes del sistema.

Se basa en 3 principios fundamentales:

- **API como Ciudadano de Primera Clase:** Las APIs se consideran elementos centrales en la arquitectura del software, no simples complementos.



- **Desarrollo Guiado por Contrato:** La especificación de la API actúa como un contrato que define cómo deben interactuar los componentes del sistema, facilitando la colaboración entre equipos.
- **Desacoplamiento de Componentes:** Al establecer interfaces claras, se permite que los equipos de frontend y backend trabajen de manera independiente, promoviendo el desarrollo paralelo.

Por otro lado, el enfoque **Code First**, también conocido como Implementation First, **prioriza la escritura del código de la aplicación antes de definir formalmente la API**. La documentación y especificación de la API se generan a partir del código existente.

Se basa en 2 principios fundamentales:

- **Desarrollo Rápido:** Se enfoca en la implementación inmediata de funcionalidades, permitiendo una rápida iteración y prototipado.
- **Flexibilidad:** Facilita cambios y ajustes durante el desarrollo, adaptándose a requisitos que evolucionan.

Aspecto	<i>API First</i>	<i>Code First</i>
<b>Filosofía</b>	Basado en el diseño previo de la API.	Basado en la implementación y evolución del código.
<b>Proceso</b>	Permite desarrollo paralelo y define interfaces claras.	Desarrollo secuencial que puede generar inconsistencias.
<b>Proceso</b>	Requiere planificación, pero otorga mayor control y escalabilidad.	Favorece la rapidez, ideal para requisitos ambiguos.
<b>Integración</b>	Facilita la conexión con sistemas externos.	Puede complicar la integración si la API está muy acoplada.



## Bases de datos relacionales

El **modelo relacional** fue introducido por Edgar F. Codd en 1970 mediante su publicación "A Relational Model of Data for Large Shared Data Banks". Codd propuso **representar los datos mediante relaciones (tablas)**, donde cada fila (**tupla**) corresponde a un registro y cada columna (**atributo**) a un dato específico. Este enfoque matemático, basado en la teoría de conjuntos y la lógica de predicados, ofrecía una estructura más flexible y coherente en comparación con los modelos jerárquicos y de red predominantes en la época.

A pesar de la reticencia inicial de IBM para adoptar el modelo, su potencial fue rápidamente reconocido por otros actores del mercado. En 1979, Oracle lanzó el primer sistema de gestión de bases de datos (SGBD) comercial basado en el modelo relacional, implementando el lenguaje SQL, desarrollado previamente en el proyecto System R de IBM. Desde entonces, el modelo relacional se ha consolidado como el estándar en la gestión de datos, siendo la base de numerosos sistemas y aplicaciones en diversas áreas.

Para garantizar que un sistema de gestión de bases de datos se adhiera fielmente al modelo relacional, Codd formuló un conjunto de 12 reglas, numeradas del 0 al 12. Entre estas reglas se encuentran criterios a destacar, como:

- **Representación de la información:** Toda la información debe estar representada explícitamente mediante valores en tablas.
- **Acceso garantizado:** Cada dato debe ser accesible mediante una combinación de nombre de tabla, clave primaria y nombre de columna.
- **Tratamiento sistemático de valores nulos:** El sistema debe manejar los valores nulos de manera uniforme y coherente.
- **Catálogo en línea basado en el modelo relacional:** La estructura de la base de datos debe estar representada en el mismo modelo relacional, permitiendo su consulta mediante el lenguaje estándar.

El diseño efectivo de una base de datos relacional comienza con un modelado conceptual adecuado. En 1976, Peter Chen propuso el **modelo Entidad-Relación (E/R)** como una herramienta para representar de manera gráfica y estructurada los datos y sus interrelaciones. Este modelo introduce conceptos como entidades (objetos del mundo real), atributos (propiedades de las entidades) y relaciones (asociaciones entre entidades), facilitando la comprensión y el diseño de la base de datos.



Para gestionar la complejidad y promover la independencia de los datos, que esto suponía, se adoptó la **arquitectura de tres niveles** propuesta por ANSI/SPARC en 1975:

- **Nivel interno:** Describe la organización física de los datos en el almacenamiento.
- **Nivel conceptual:** Representa la estructura lógica de la base de datos, integrando las diferentes vistas externas.
- **Nivel externo:** Define las diferentes vistas de los usuarios, adaptadas a sus necesidades específicas.

Esta separación permite que los cambios en un nivel no afecten directamente a los otros, facilitando la evolución y mantenimiento del sistema.

La **normalización** es un proceso sistemático para organizar los datos en una base de datos, con el objetivo de reducir la redundancia y mejorar la integridad de los datos. Codd introdujo las primeras formas normales, y posteriormente, junto con otros investigadores, se desarrollaron formas adicionales como la Forma Normal de Boyce-Codd (FNBC), cuarta y quinta formas normales.

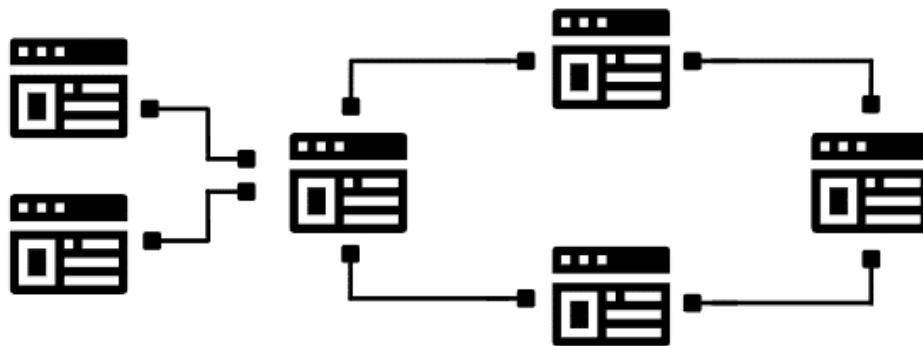
Cada forma normal establece criterios específicos para la organización de los datos:

- **Primera Forma Normal (1FN):** Elimina grupos repetitivos, asegurando que cada campo contenga solo un valor atómico.
- **Segunda Forma Normal (2FN):** Elimina dependencias parciales, asegurando que todos los atributos dependan completamente de la clave primaria.
- **Tercera Forma Normal (3FN):** Elimina dependencias transitivas, garantizando que los atributos no clave dependan únicamente de la clave primaria.
- **Forma Normal de Boyce-Codd (FNBC):** Refina la 3FN, abordando ciertos casos especiales de dependencias.

Los **SGBD** son herramientas esenciales que permiten la creación, manipulación y administración de bases de datos. Proporcionan interfaces para definir estructuras de datos, realizar consultas, garantizar la seguridad y mantener la integridad de los datos. Entre los SGBD más destacados se encuentran Oracle, MySQL, PostgreSQL y Microsoft SQL Server, cada uno con características específicas que los hacen adecuados para diferentes contextos y necesidades.



**El lenguaje SQL (Structured Query Language) es el estándar para la interacción con bases de datos relacionales.** Permite definir estructuras de datos (DDL), manipular datos (DML) y controlar el acceso a la base de datos (DCL). Su sintaxis declarativa facilita la formulación de consultas complejas y su integración en diversas aplicaciones.



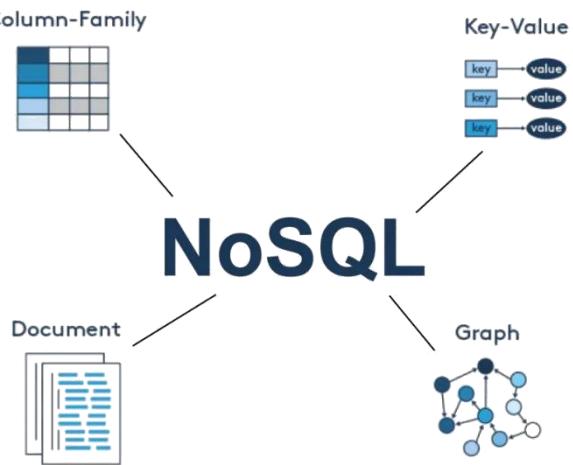
### Bases de datos NoSQL

Las bases de datos NoSQL (del inglés **Not Only SQL**) emergieron como respuesta a las limitaciones de los sistemas de gestión de bases de datos relacionales (RDBMS) tradicionales, especialmente en escenarios que requieren alta escalabilidad, flexibilidad en el modelado de datos y tolerancia a fallos. El término "NoSQL" fue acuñado en 1998, pero su adopción masiva se produjo a partir de la década de 2000, impulsada por las necesidades de grandes empresas tecnológicas como Facebook, Amazon, LinkedIn y Google, que enfrentaban desafíos en el manejo de grandes volúmenes de datos y estructuras complejas.

Las bases de datos NoSQL se clasifican en varias categorías, cada una adecuada para diferentes tipos de aplicaciones:

- **Bases de datos de documentos:** Almacenan datos en formatos como JSON o BSON. Ejemplo: MongoDB.
- **Bases de datos clave-valor:** Utilizan pares clave-valor para almacenar datos. Ejemplo: Redis.
- **Bases de datos de columnas:** Almacenan datos en columnas en lugar de filas, optimizando consultas sobre grandes volúmenes de datos. Ejemplo: Apache Cassandra.

- **Bases de datos de grafos:** Representan datos en nodos y relaciones, ideales para modelar redes y relaciones complejas. Ejemplo: Neo4j.



Las bases de datos NoSQL ofrecen varias ventajas sobre los sistemas relacionales:

- **Escalabilidad horizontal:** Permiten distribuir datos en múltiples nodos, facilitando el manejo de grandes volúmenes de información.
- **Flexibilidad en el esquema:** No requieren una estructura de datos fija, lo que permite adaptarse rápidamente a cambios en los requisitos de la aplicación.
- **Alto rendimiento:** Optimizadas para operaciones de lectura y escritura rápidas, especialmente en aplicaciones web y móviles.

A pesar de sus ventajas, las bases de datos NoSQL presentan ciertos desafíos:

- **Falta de estandarización:** La ausencia de un lenguaje de consulta universal dificulta la interoperabilidad entre diferentes sistemas.
- **Consistencia eventual:** Muchas bases de datos NoSQL optan por una consistencia eventual en lugar de una consistencia fuerte, lo que puede complicar el desarrollo de aplicaciones que requieren transacciones estrictas.

## Transacciones y atomicidad en las bases de datos

Las transacciones constituyen un elemento fundamental para garantizar la integridad y coherencia de los datos en los sistemas de gestión de bases de datos. **Una transacción se define como una unidad lógica de trabajo que agrupa una o varias operaciones que deben ejecutarse de manera completa y sin errores para mantener la consistencia del sistema.** Estas operaciones pueden incluir inserciones, actualizaciones, eliminaciones o consultas, y su correcta ejecución es esencial para preservar la integridad de la base de datos.

El comportamiento de las transacciones se rige por las propiedades ACID: Atomicidad, Consistencia, Aislamiento y Durabilidad. Estas propiedades fueron formalizadas por Theo Härder y Andreas Reuter en 1983, basándose en trabajos previos de Jim Gray, quien ya había identificado la atomicidad, consistencia y durabilidad como características esenciales de las transacciones.

- La **atomicidad** es una propiedad crítica que garantiza que todas las operaciones dentro de una transacción se completen con éxito o que ninguna de ellas tenga efecto. Este principio de "todo o nada" es fundamental para mantener la coherencia de la base de datos.

Para implementar la atomicidad, los SGBD utilizan mecanismos como el registro de transacciones (write-ahead logging), donde todas las operaciones se registran antes de aplicarse, permitiendo revertir los cambios en caso de fallo. Otra técnica es la paginación de sombra, que crea una copia de las páginas modificadas y solo las reemplaza en el almacenamiento principal una vez que la transacción se ha completado con éxito.

- El **aislamiento** es esencial para el control de concurrencia en sistemas donde múltiples transacciones pueden ejecutarse simultáneamente. Para gestionar esto, se emplean niveles de aislamiento como Lectura No Confirmada, Lectura Confirmada, Lectura Repetible y Serializable, cada uno con diferentes garantías sobre la visibilidad de los cambios realizados por otras transacciones.

Además, se utilizan técnicas como el Control de Concurrencia Optimista (**OCC**) y el Control de Concurrencia mediante Versiones Múltiples (**MVCC**). El OCC permite que las transacciones se ejecuten sin bloqueos, verificando al final si ha habido conflictos, mientras que el MVCC mantiene múltiples versiones de los datos para permitir lecturas consistentes sin bloquear escrituras.



- La **durabilidad** asegura que los cambios realizados por una transacción confirmada persistan en el sistema, incluso en caso de fallos. Esto se logra mediante técnicas como el registro de transacciones y la escritura diferida, donde los cambios se almacenan en medios persistentes antes de considerarse definitivos.

En caso de fallos, los SGBD utilizan los registros de transacciones para recuperar el estado consistente de la base de datos, aplicando o deshaciendo las operaciones según sea necesario. Este proceso es fundamental para mantener la integridad de los datos y la confianza en el sistema.



### 3. TEMPORALIZACIÓN

En este apartado se detalla la planificación y ejecución de las siete semanas de trabajo, estructuradas en cinco sprints coincidentes con las fases principales del proyecto: diario de planificación, análisis, investigación, diseño y codificación. Cada sprint abarcó aproximadamente una semana, salvo el sprint de diseño y codificación que requirió dos semanas dada su complejidad práctica. No se llevaron a cabo reuniones diarias formales (daily scrum), ya que el proyecto fue realizado de forma individual, optándose por un seguimiento mixto de Scrum y Kanban adaptado a un desarrollador único.

#### 3.1. Sprint 0: Diario de planificación

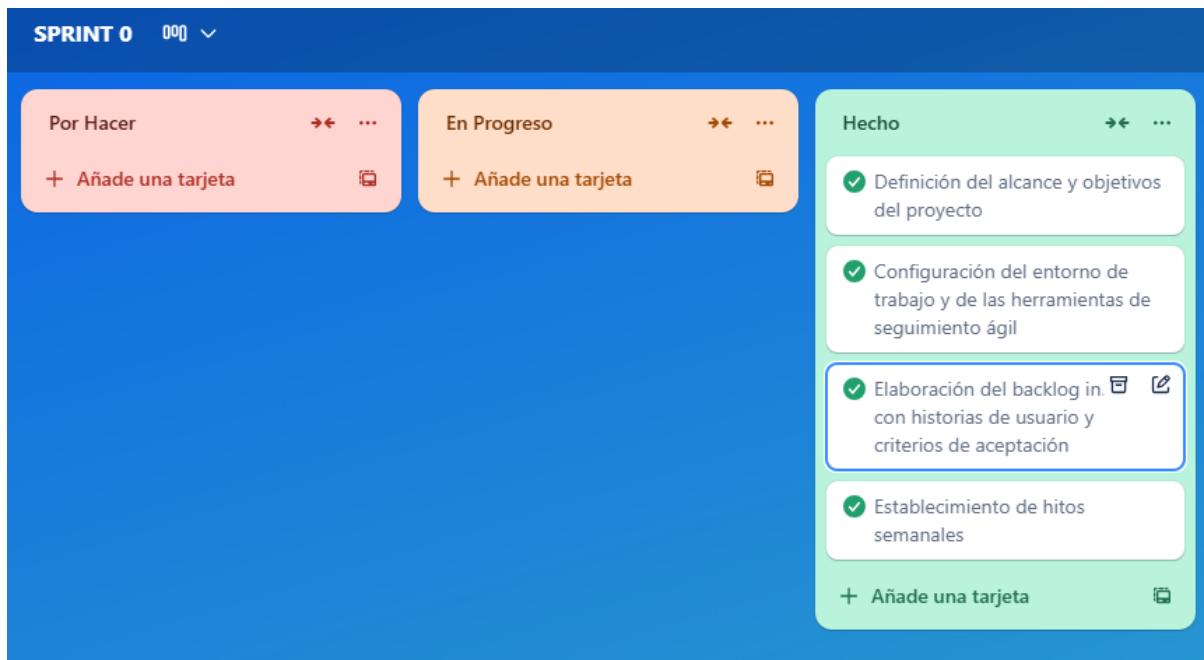
**Duración:** Semana 1

Durante la primera semana se estableció el diario de planificación, fundamentado en una metodología ágil híbrida entre Scrum y Kanban. Se definieron los objetivos globales y se creó un tablero Kanban con las columnas “*Por hacer*”, “*En progreso*” y “*Hecho*”. A diario se revisaron las prioridades y se ajustaron las tareas mediante tickets en la herramienta seleccionada (Trello). Aunque no se convocaron dailys formales, se reservó media hora cada día para revisar el tablero y replanificar según los avances y obstáculos detectados. Además, se identificaron los criterios de éxito, los entregables esperados y las métricas de rendimiento.

**Tareas clave:**

- Definición del alcance y objetivos del proyecto.
- Configuración del entorno de trabajo y de las herramientas de seguimiento ágil.
- Elaboración del backlog inicial con historias de usuario y criterios de aceptación.
- Establecimiento de hitos semanales.





### 3.2. Sprint 1: Análisis

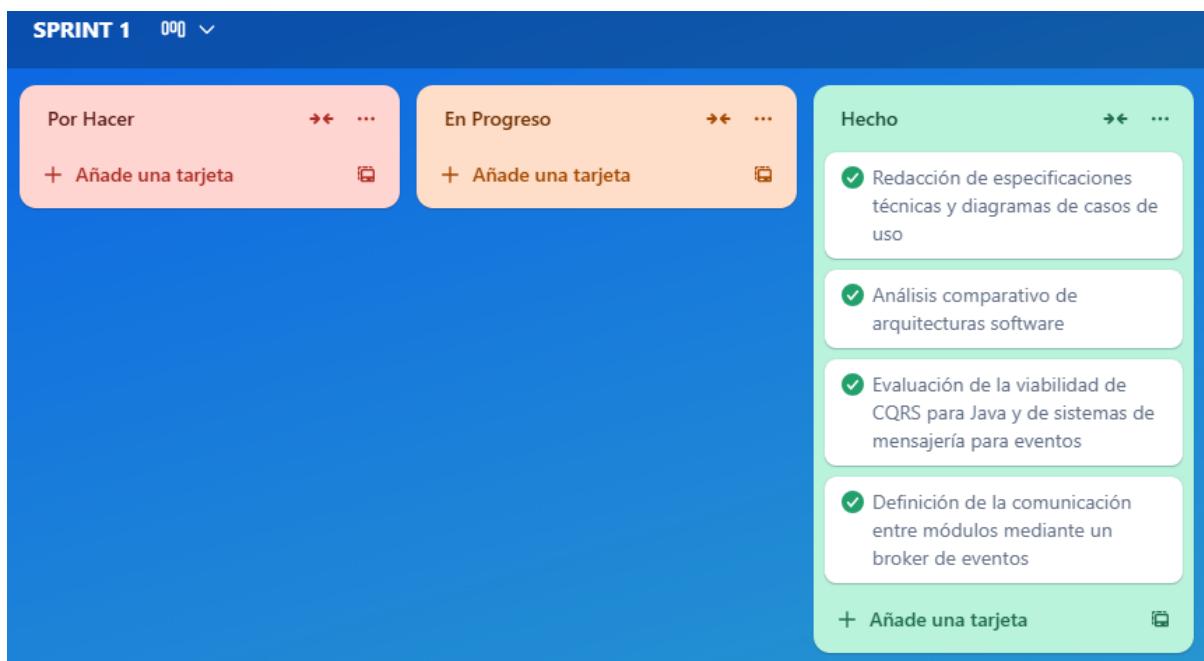
**Duración:** Semana 2

El segundo sprint se centró en la fase de análisis, cuyo propósito fue determinar con precisión qué se quería desarrollar y cómo implementarlo. Se revisaron los requisitos funcionales y no funcionales, se recogieron las necesidades de los usuarios potenciales y se evaluaron las restricciones técnicas y de negocio. También se exploraron distintas arquitecturas (monolítica, hexagonal, microservicios) y se seleccionó el patrón híbrido basado en eventos para desacoplar el módulo de autenticación en Go y el de gestión de cursos en Java.

#### Actividades desarrolladas:

- Redacción de especificaciones técnicas y diagramas de casos de uso.
- Análisis comparativo de arquitecturas software (hexagonal vs. microservicios).
- Evaluación de la viabilidad de CQRS para Java y de sistemas de mensajería para eventos.
- Definición de la comunicación entre módulos mediante un broker de eventos.





### 3.3. Sprint 2: Investigación

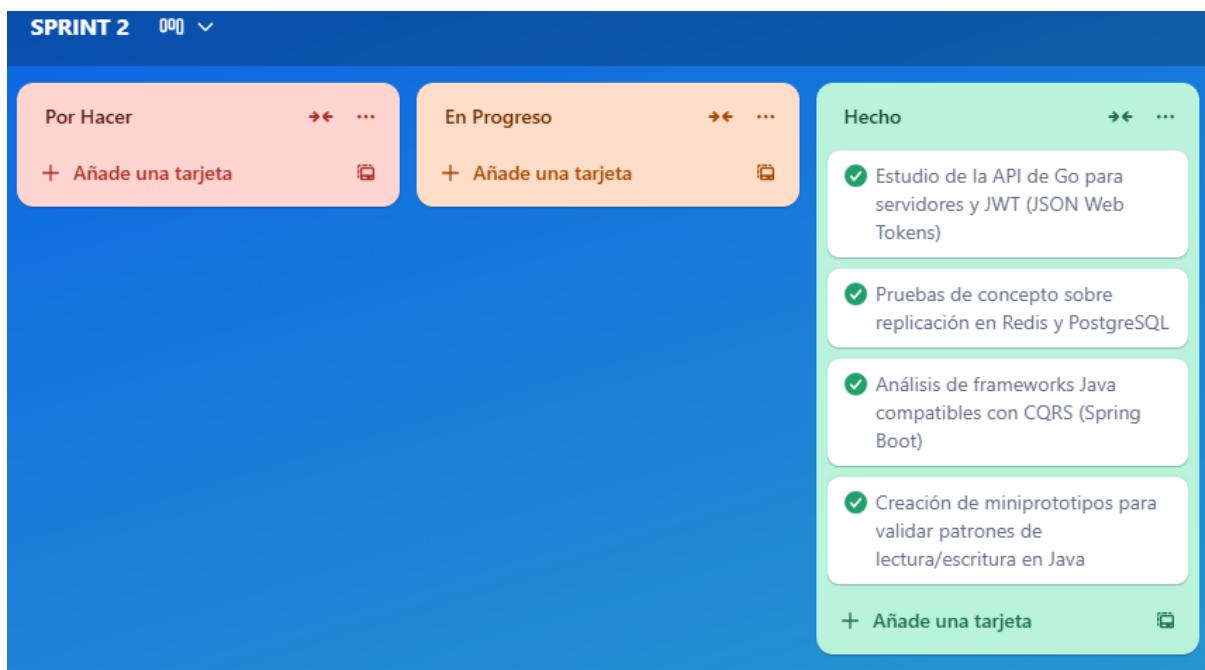
**Duración:** Semana 3

La fase de investigación tuvo como objetivo adquirir los conocimientos necesarios para abordar la implementación. Se profundizó en el aprendizaje del lenguaje Go, en la integración de PostgreSQL y Redis, así como en el patrón CQRS en entornos Java. Se consultaron documentación oficial, artículos académicos y tutoriales de buenas prácticas.

**Actividades desarrolladas:**

- Estudio de la API de Go para servidores web y JWT (JSON Web Tokens).
- Pruebas de concepto sobre replicación en Redis y PostgreSQL.
- Análisis de frameworks Java compatibles con CQRS (Spring Boot).
- Creación de miniprototipos para validar patrones de lectura/escritura en Java.





### 3.4. Sprint 3-4: Diseño

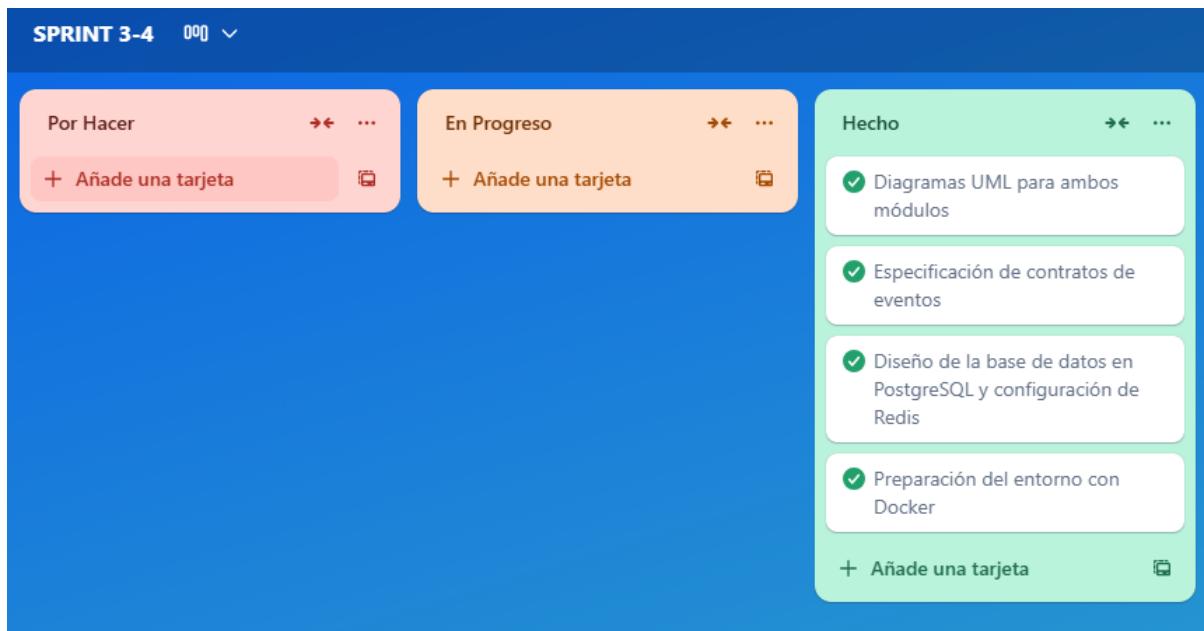
**Duración:** Semanas 4 y 5

En estas dos semanas se ejecutó la fase de diseño, la parte más práctica donde se aplicaron los conocimientos adquiridos. Se elaboraron diagramas UML detallados, se definieron las entidades y los repositorios, y se diseñaron las interfaces de eventos. Asimismo, se diseñó la capa de servicios y la separación entre comandos y consultas, así como la topología de la base de datos relacional y la caché en Redis.

**Entregables principales:**

- Diagramas UML para ambos módulos.
- Especificación de contratos de eventos.
- Diseño de la base de datos en PostgreSQL y configuración de Redis.
- Preparación del entorno con Docker.





### 3.5. Sprint 5-6: Codificación

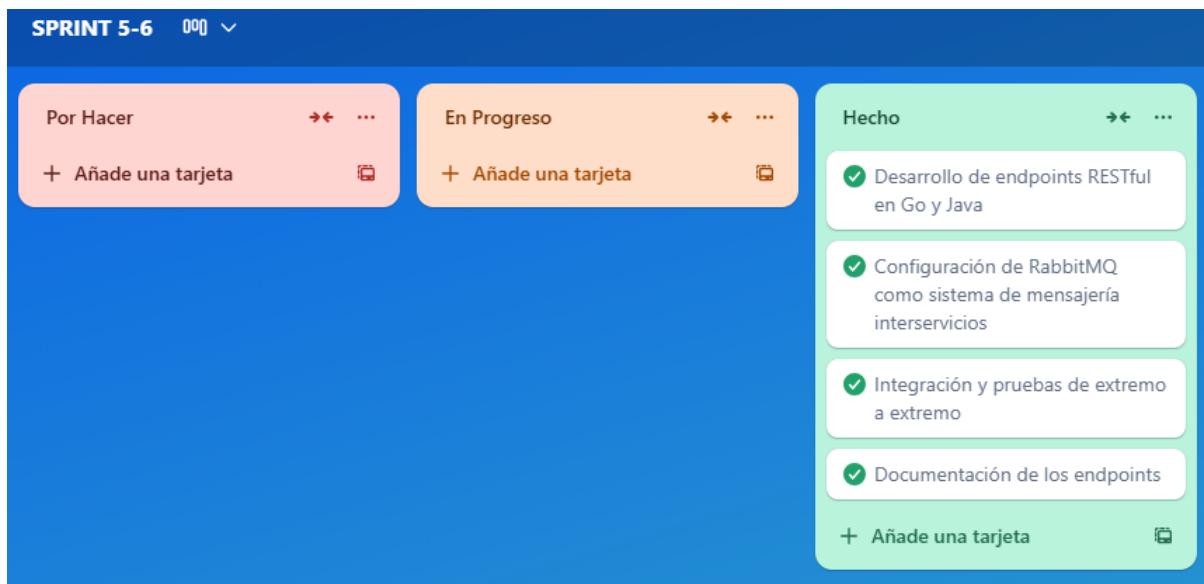
**Duración:** Semanas 6 y 7

Finalmente, el desarrollo de la codificación se llevó a cabo en las últimas dos semanas, extendiéndose ligeramente para ajustes y pruebas. Se implementó el módulo de autenticación en Go, configurando rutas, middleware de seguridad y generación de tokens; y el módulo de gestión de cursos en Java, aplicando CQRS para separar comandos y consultas. Se integró la comunicación por eventos usando RabbitMQ, se implementaron repositorios para PostgreSQL y adaptadores para Redis, y se añadieron pruebas.

**Tareas realizadas:**

- Desarrollo de endpoints RESTful en Go y Java.
- Configuración de RabbitMQ como sistema de mensajería interservicios.
- Integración y pruebas de extremo a extremo (Go  $\leftarrow\rightarrow$  Java).
- Optimización de consultas y validación de rendimiento.
- Documentación de los endpoints.





## 4. ANÁLISIS, DISEÑO E INFRAESTRUCTURA

En este capítulo se presenta la base técnica del sistema GoLearnix, abarcando desde el análisis funcional hasta la infraestructura propuesta. Se describe cómo se modelan los requisitos y actores clave del sistema.

### 4.1. Análisis

En este bloque se profundiza en los requisitos y actores del sistema, describiendo sus interacciones mediante casos de uso.

#### 4.1.1. Requisitos iniciales

Para garantizar que se cubra las necesidades de usuarios finales y cumpla con los objetivos de escalabilidad, seguridad y mantenibilidad, se identificaron los siguientes requisitos como producto mínimo viable (MVP):

##### Requisitos funcionales:

###### Aplicación en Go - Gestión de usuarios y autenticación:

- Registro de nuevos usuarios: **POST** /api/v1/auth/register
- Inicio de sesión y emisión de token JWT: **POST** /api/v1/auth/login
- Validación de la sesión vigente: **GET** /api/v1/auth/validate
- Obtención de información del usuario a partir del token: **GET** /api/v1/user/me
- Cierre de sesión: **POST** /api/v1/auth/logout
- Eliminación de cuenta de usuario, con propagación de eventos para sincronizar datos en el módulo de cursos: **DELETE** /api/v1/user/delete

###### Aplicación en Java - Gestión de cursos:

- Consulta de todos los cursos disponibles: **GET** /api/v1/golearnix/courses
- Acceso a detalles de un curso concreto: **GET** /api/v1/golearnix/courses/{courseId}
- Rol Instructor:
  - Creación de un curso: **POST** /api/v1/golearnix/courses
  - Actualización de un curso: **PUT** /api/v1/golearnix/courses/{courseId}



- Eliminación de un curso: **DELETE** /api/v1/golearnix/courses/{courseId}
- Rol Estudiante:
  - Inscripción en un curso: **POST**  
/api/v1/golearnix/courses/{courseId}/enrollments
  - Completar una lección: **POST**  
/api/v1/golearnix/courses/{courseId}/sections/{sectionId}/lessons/{lessonId}  
/complete

#### Requisitos no funcionales:

- **Seguridad:** almacenamiento de credenciales con hashing (bcrypt), autenticación basada en JWT.
- **Escalabilidad:** diseño de microservicios desacoplados, uso de RabbitMQ para eventos asíncronos y Redis para lecturas.
- **Mantenibilidad:** separación clara entre módulos Go (autenticación) y Java (gestión de cursos y CQRS).

#### 4.1.2. Actores del sistema

El sistema reconoce cuatro tipos de usuarios, organizados en una jerarquía de privilegios:

ACTOR	DESCRIPCIÓN
<b>ADMINISTRADOR</b>	Máximos privilegios: gestión completa de usuarios, roles, cursos y parámetros globales de la plataforma.
<b>INSTRUCTOR</b>	Diseña, publica, actualiza y elimina cursos. Consulta inscripciones y progreso de estudiantes.
<b>ESTUDIANTE</b>	Se inscribe en cursos, accede al material, completa lecciones y revisa su propio progreso.
<b>USUARIO NO LOGUEADO</b>	Acceso anónimo: puede explorar catálogo de cursos y sus detalles, pero no puede interactuar con elementos protegidos.

*Jerarquía de roles: cada rol hereda los permisos de los niveles inferiores (por ejemplo, un Administrador también puede inscribirse y completar lecciones como Estudiante).*

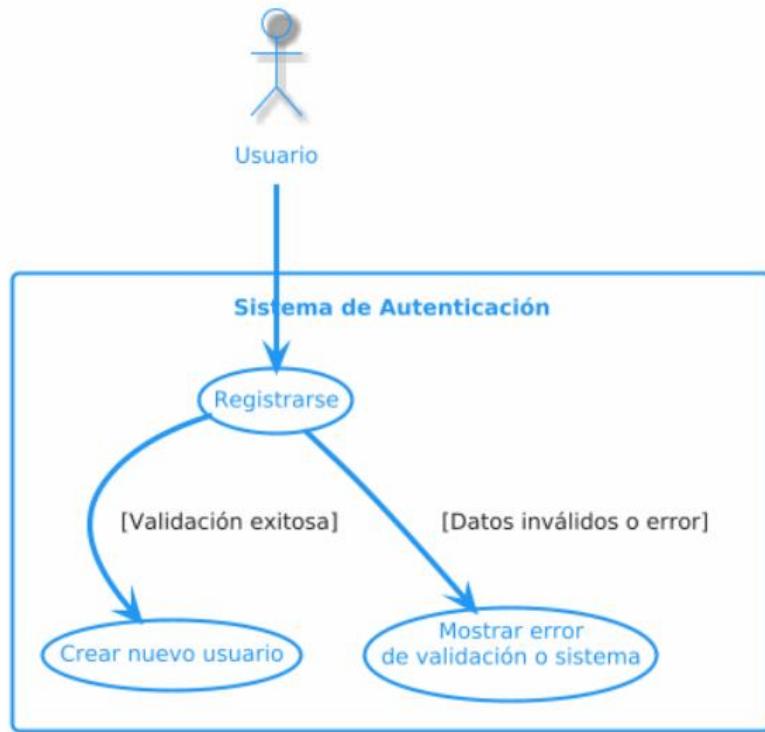


#### 4.1.3. Diagramas de casos de uso

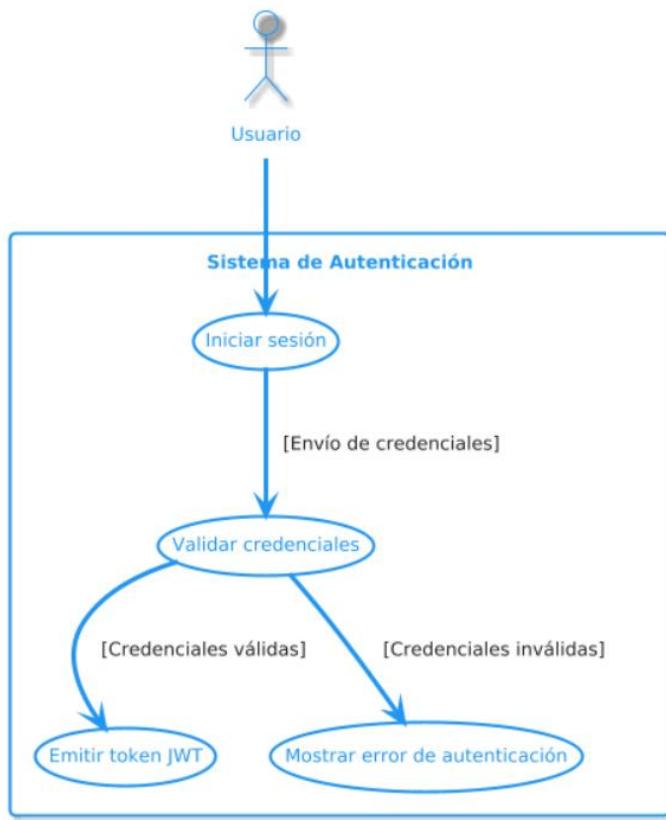
A continuación, se presentan los diagramas de casos de uso separados por cada microservicio:

##### Aplicación en Go - Gestión de usuarios y autenticación:

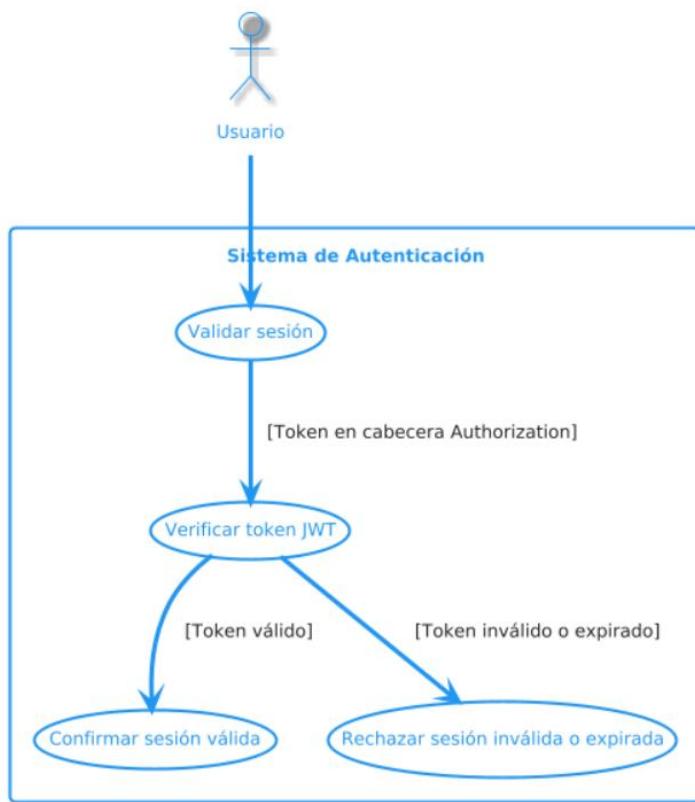
- Registro de nuevos usuarios:



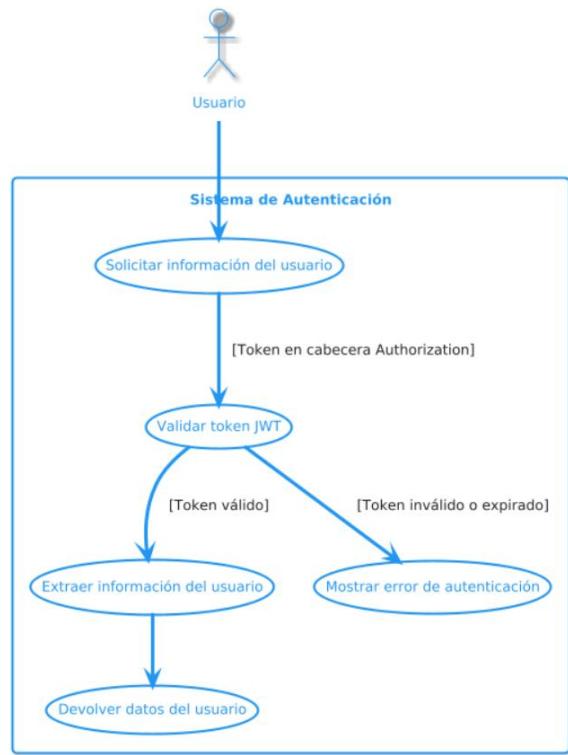
- Inicio de sesión y emisión de token JWT:



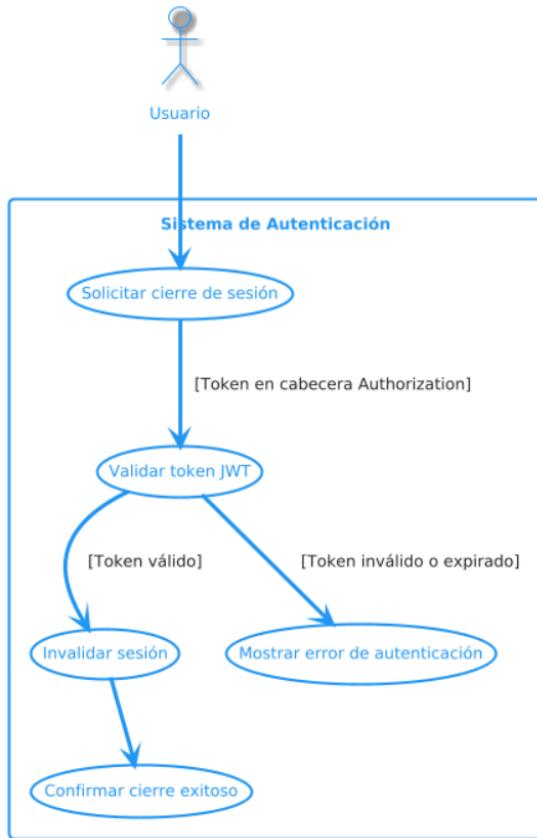
- Validación de la sesión vigente:



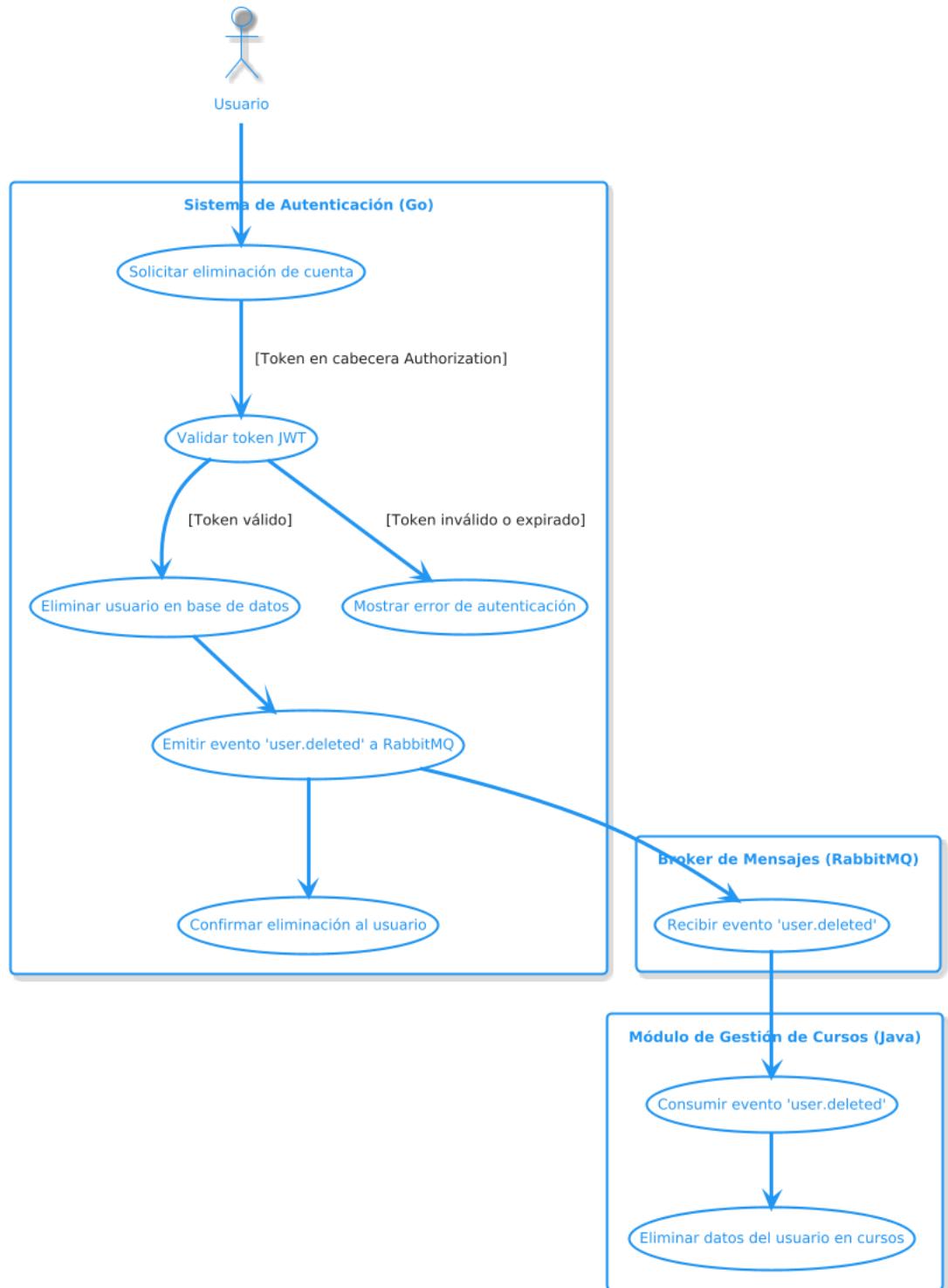
- Obtención de información del usuario a partir del token:



- Cierre de sesión:

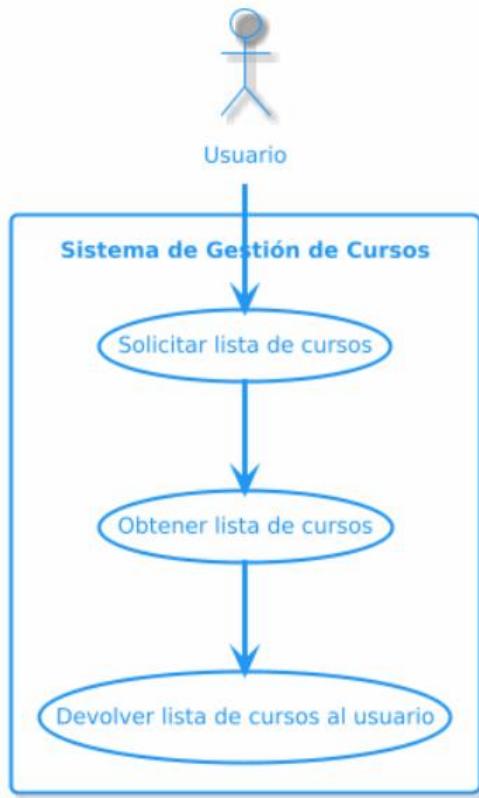


- Eliminación de cuenta de usuario, con propagación de eventos para sincronizar datos en el módulo de cursos:

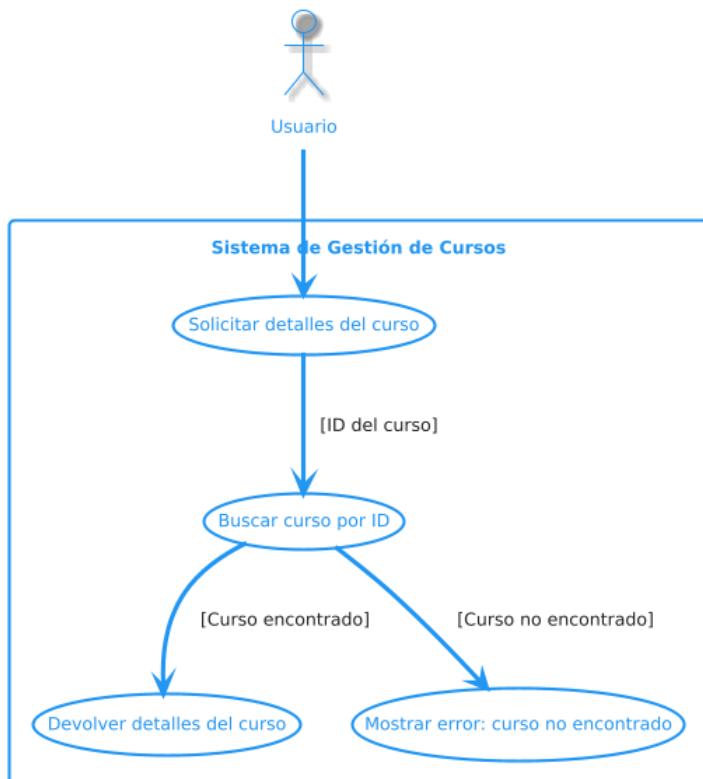


### Aplicación en Java - Gestión de cursos:

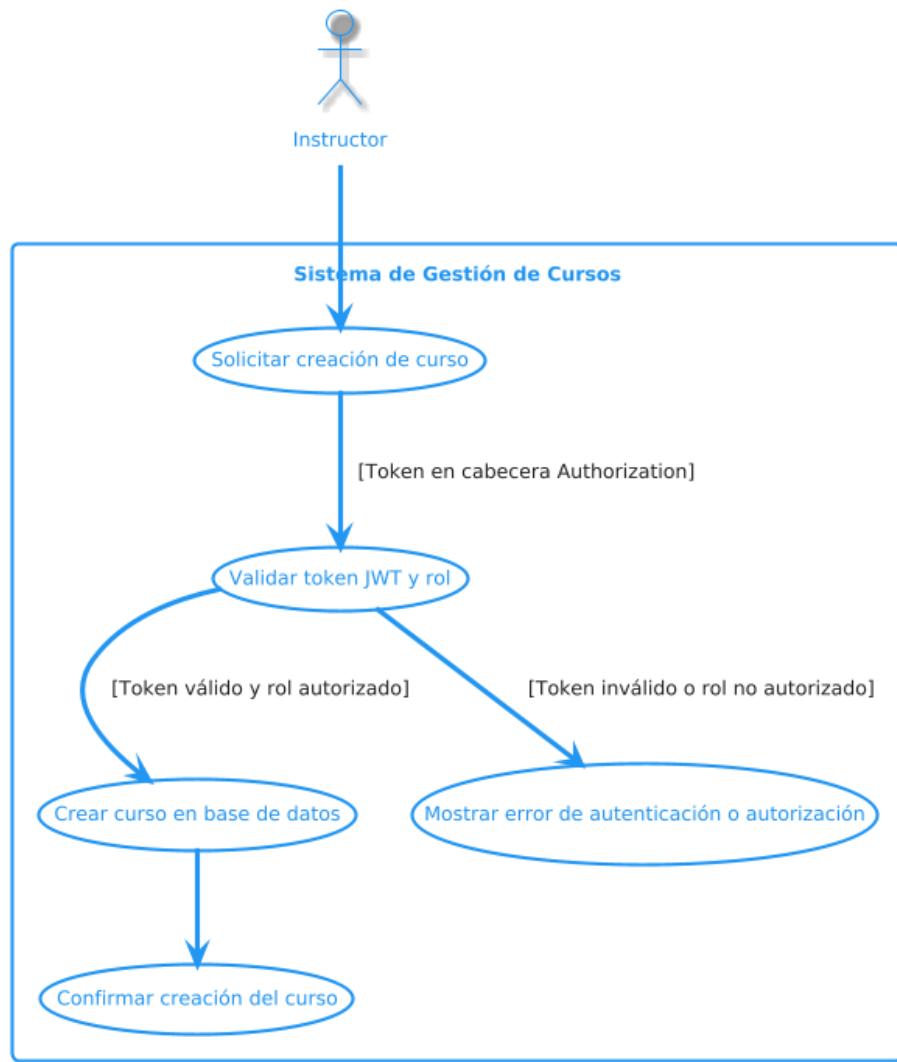
- Consulta de todos los cursos disponibles:



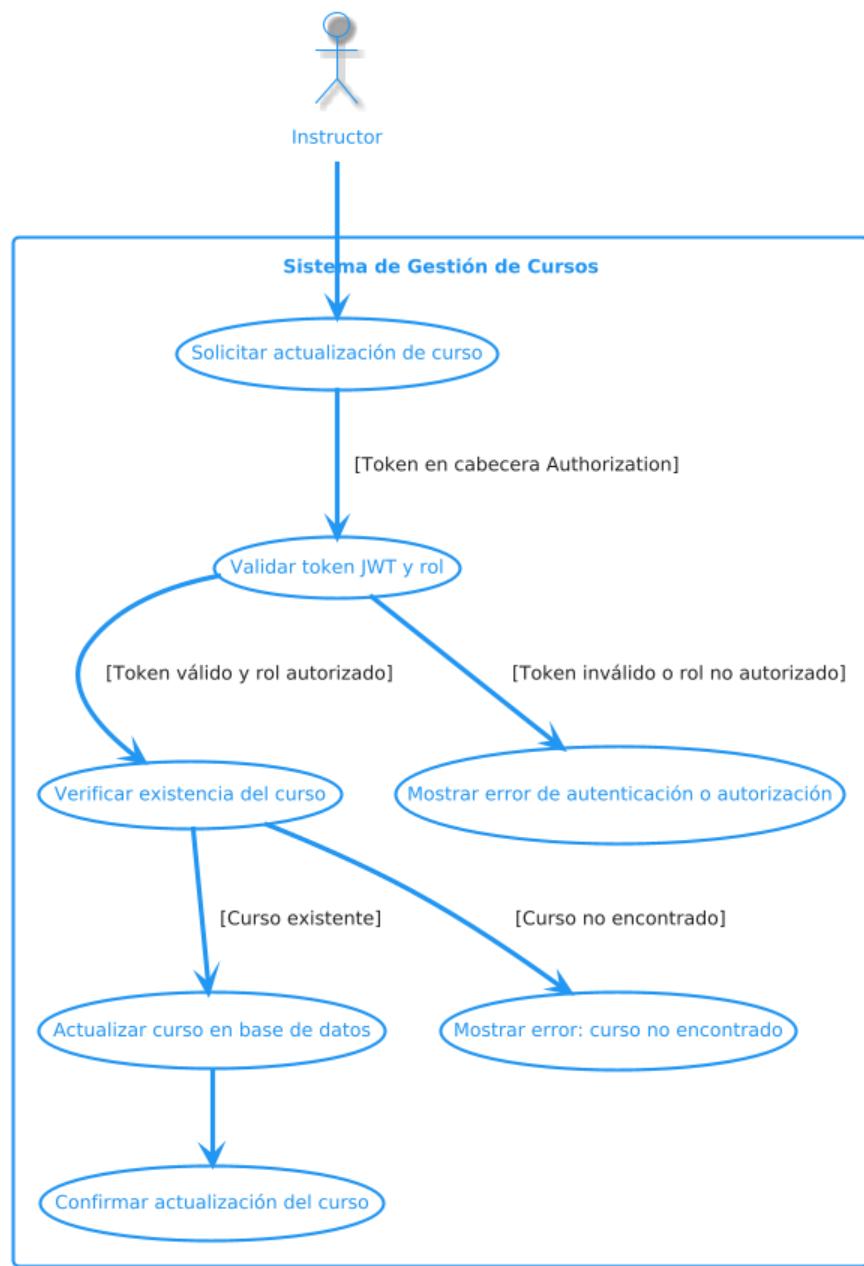
- Acceso a detalles de un curso concreto:



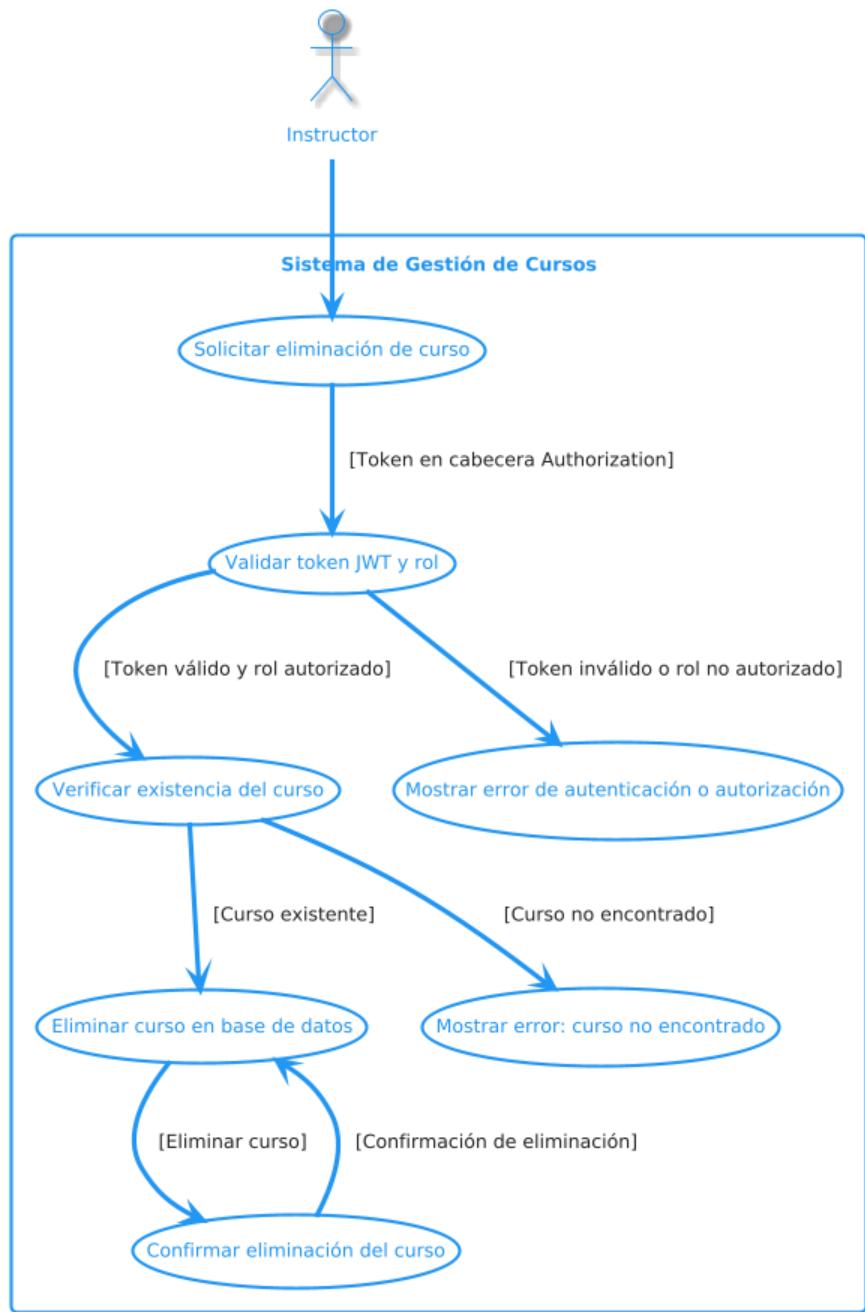
- Creación de un curso:



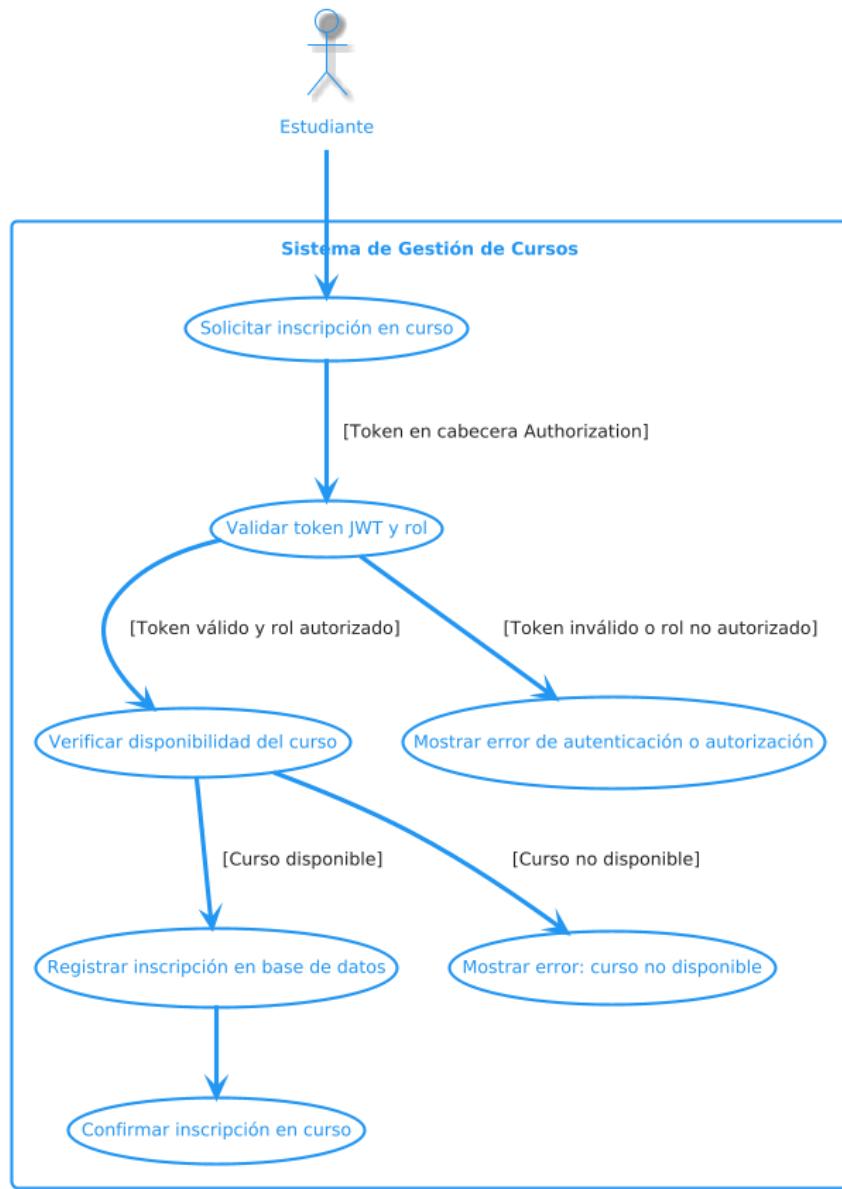
- Actualización de un curso:



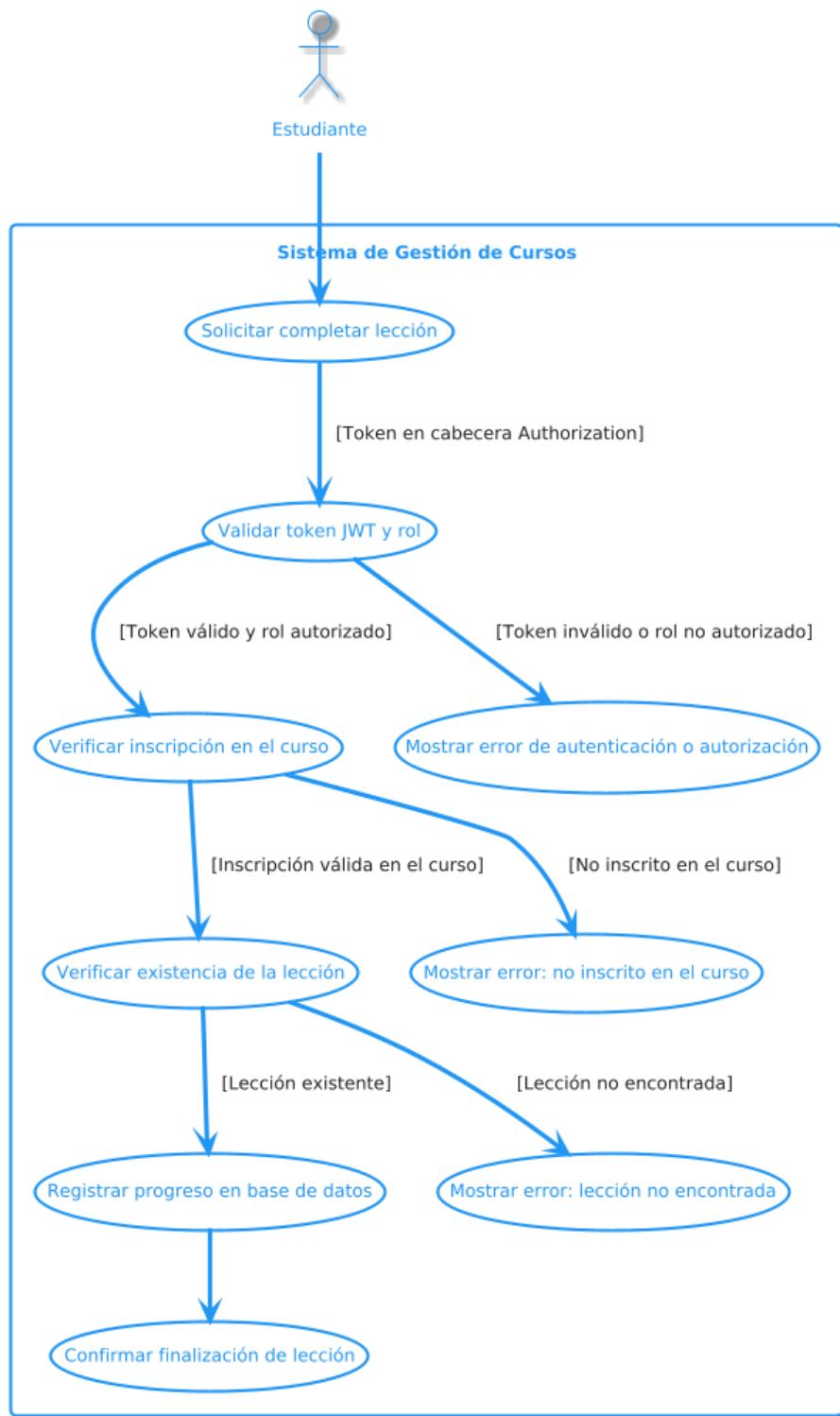
- Eliminación de un curso:



- Inscripción en un curso:



- Completar una lección:



## 4.2. Diseño

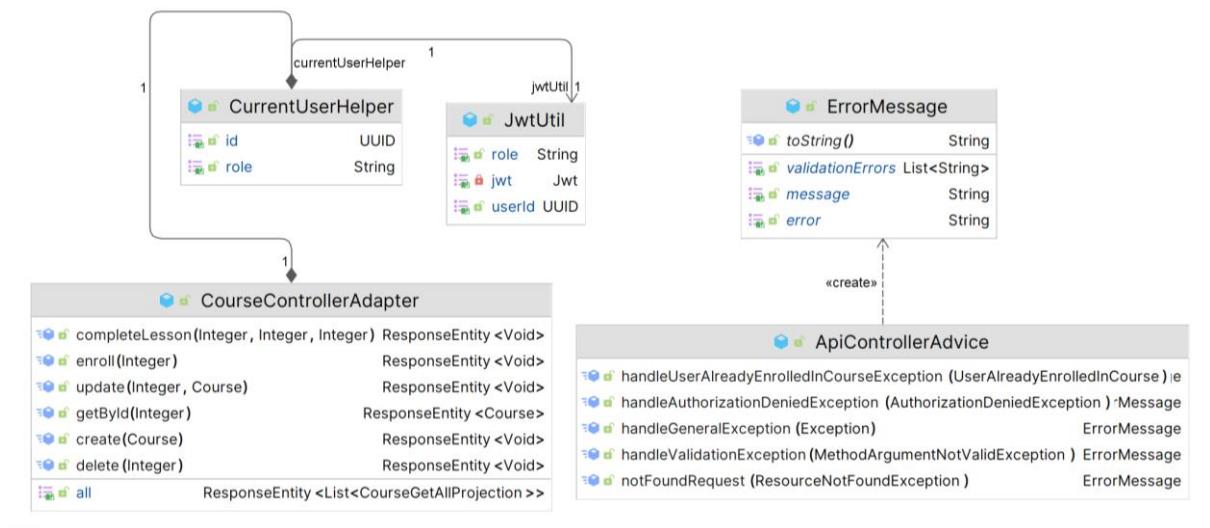
En este bloque se detallan las decisiones arquitectónicas y técnicas que estructuran el sistema GoLearnix, incluyendo diagramas de clases, secuencia y modelado de datos en persistencia.

### 4.2.1. Diagrama de clases

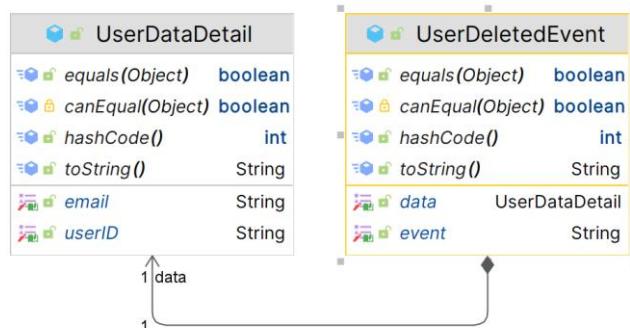
#### Aplicación en Java - Gestión de cursos:

Dado el alcance del proyecto, se ha optado por dividir las imágenes correspondientes a los diagramas de clases por módulos, con el objetivo de facilitar su comprensión.

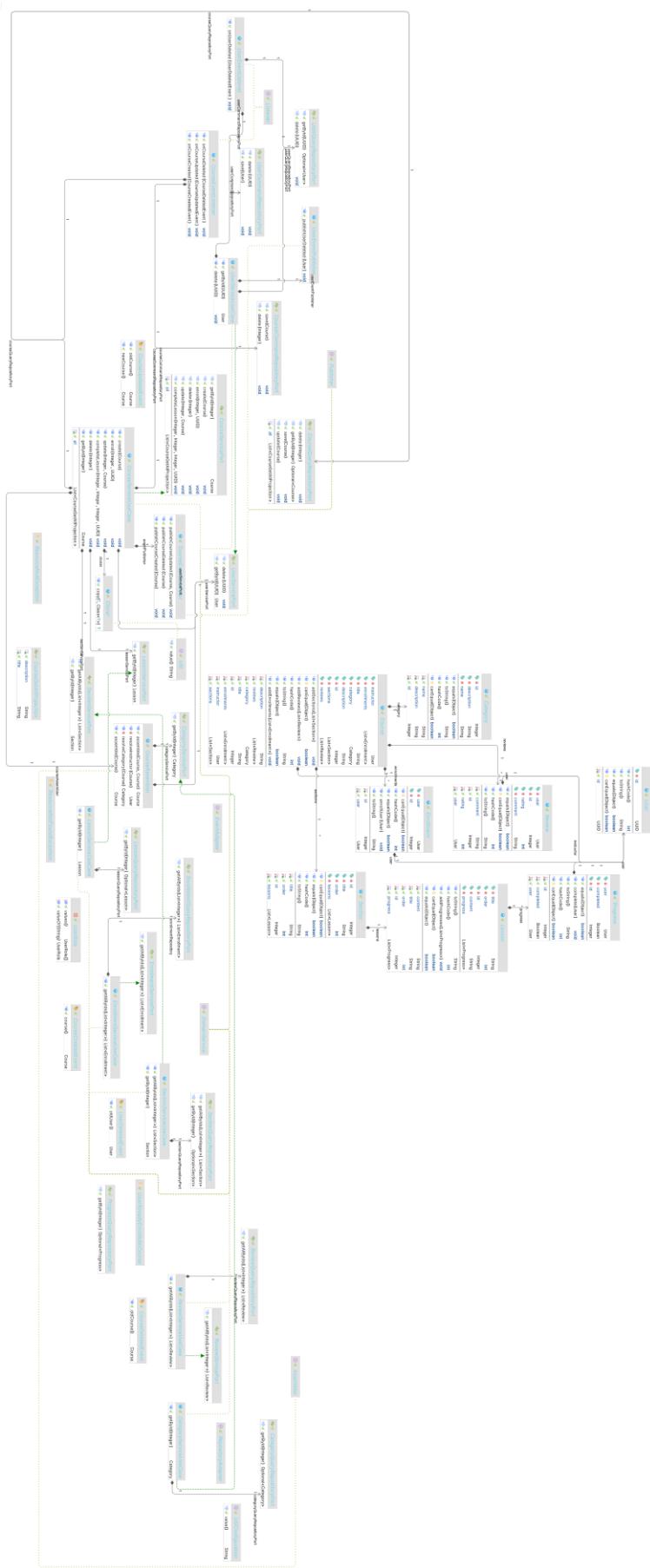
#### Módulo API-REST



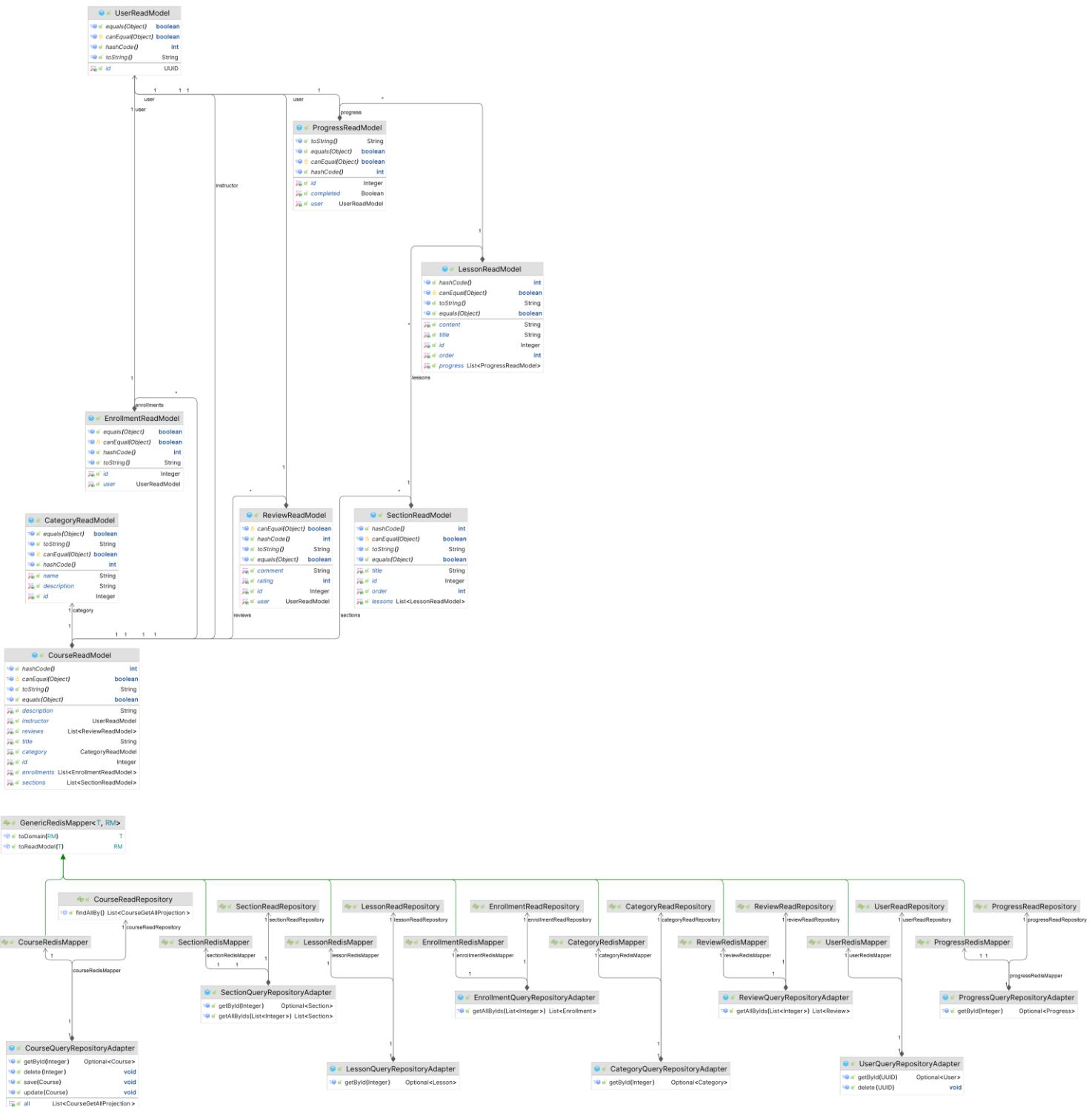
#### Módulo EVENT-RABBITMQ



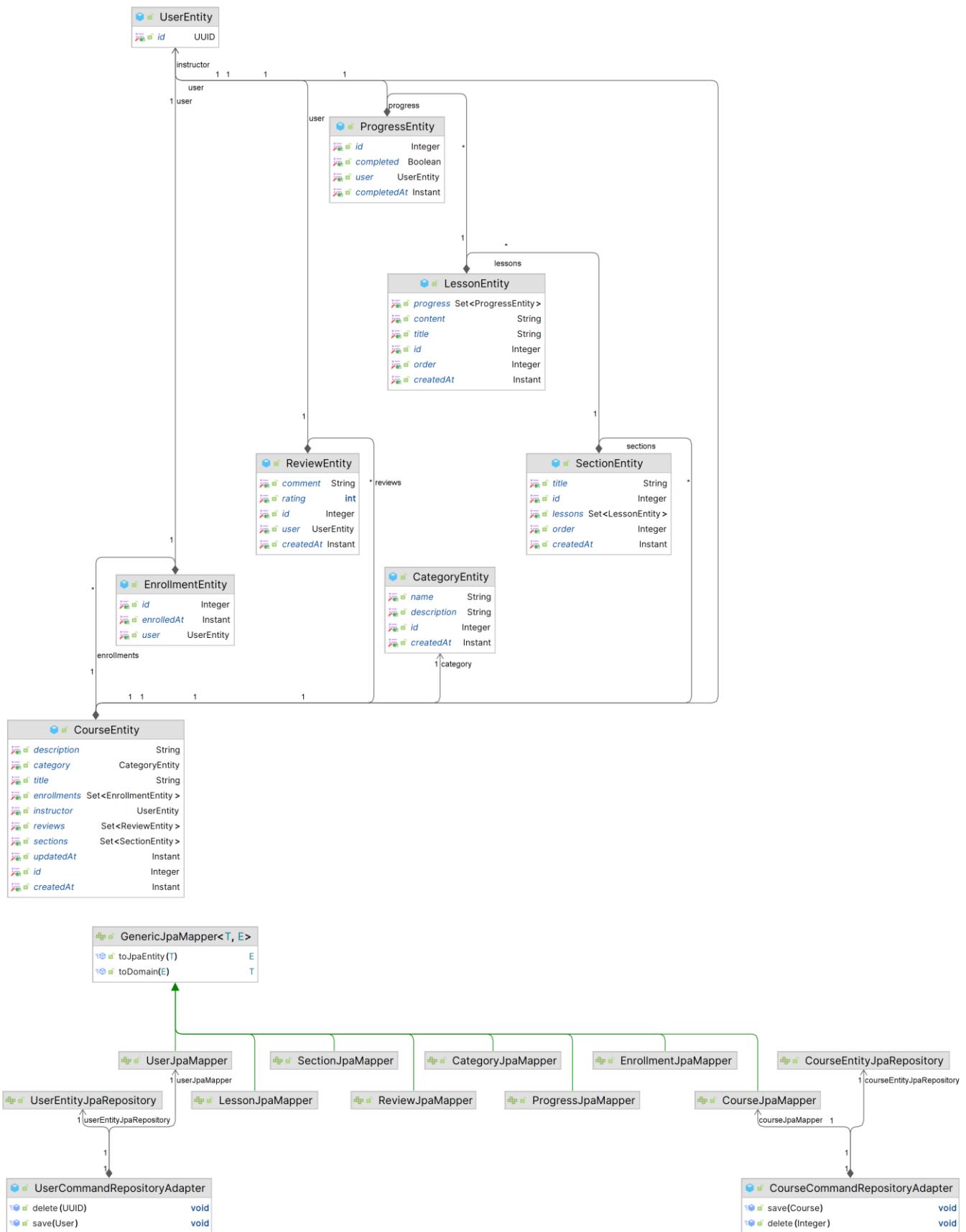
## Módulo APPLICATION



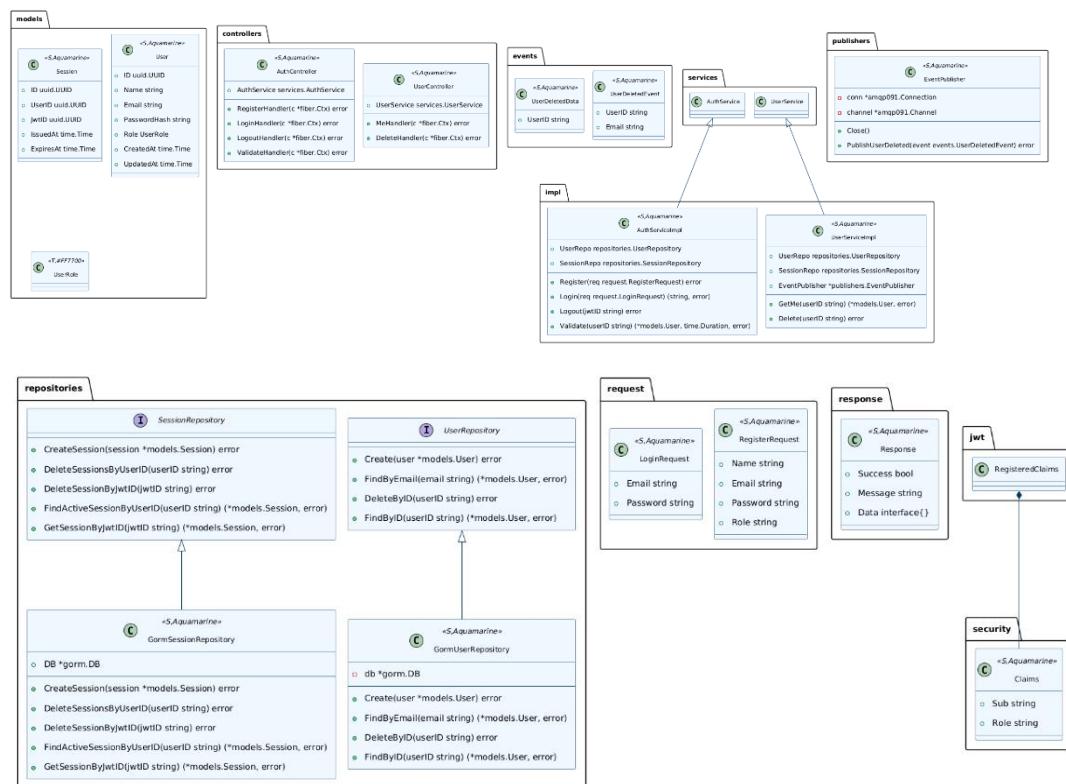
## Módulo REDIS-QUERY-REPOSITORY



## Módulo POSTGRESQL-COMMAND-REPOSITORY



### Aplicación en Go - Gestión de usuarios y autenticación:

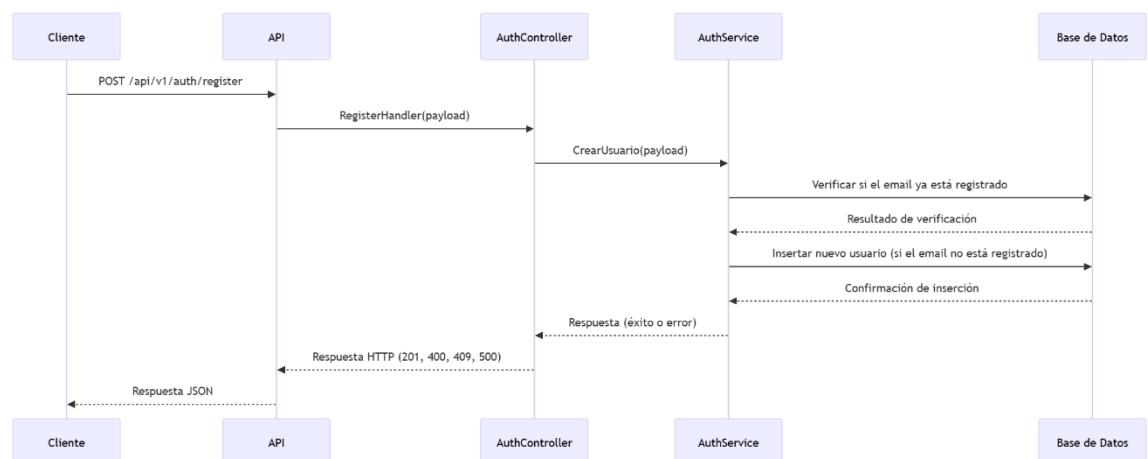


#### 4.2.2. Diagrama de secuencia

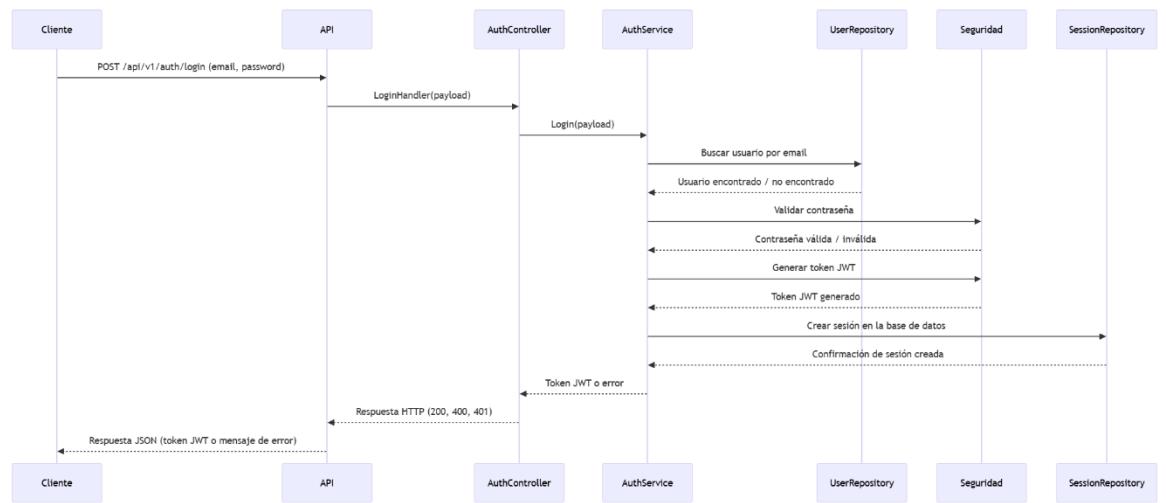
Seguidamente, se presentan los diagramas de secuencia divididos por caso de uso para facilitar su comprensión.

### Aplicación en Go - Gestión de usuarios y autenticación:

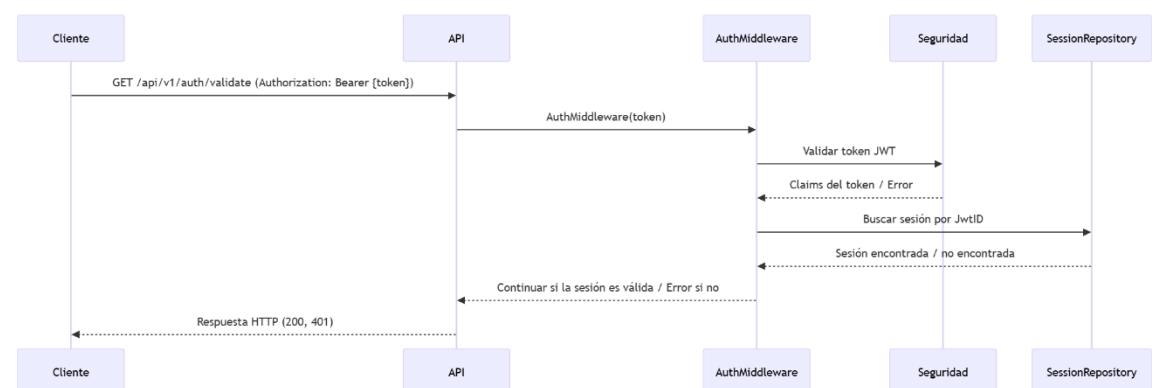
- Registro de nuevos usuarios:



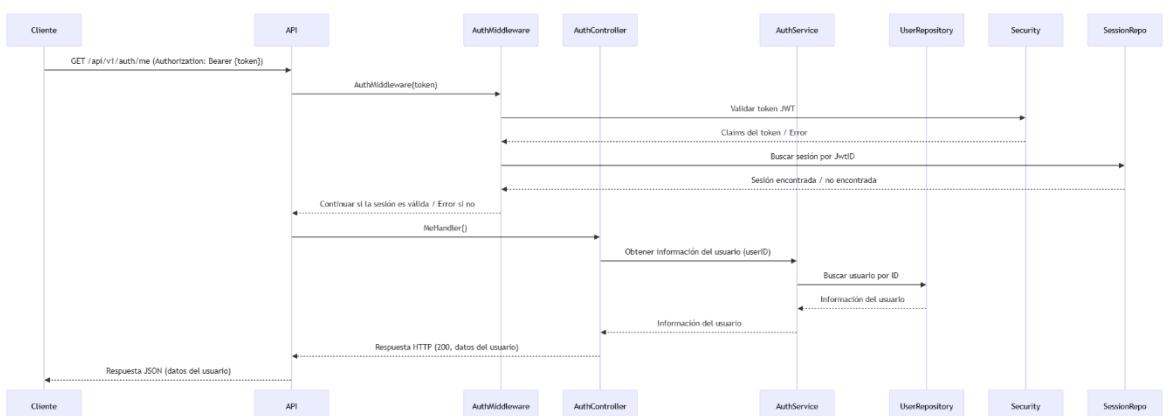
- Inicio de sesión y emisión de token JWT:



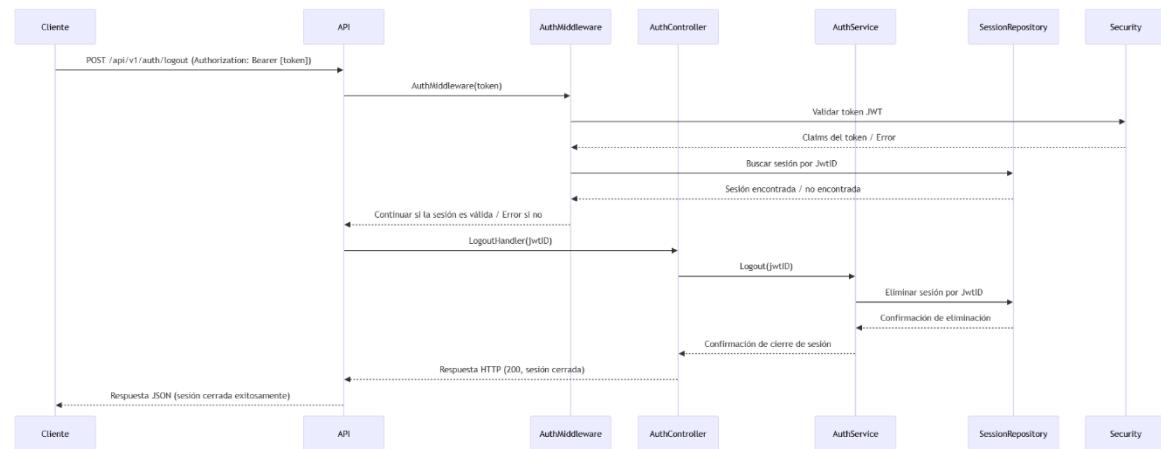
- Validación de la sesión vigente:



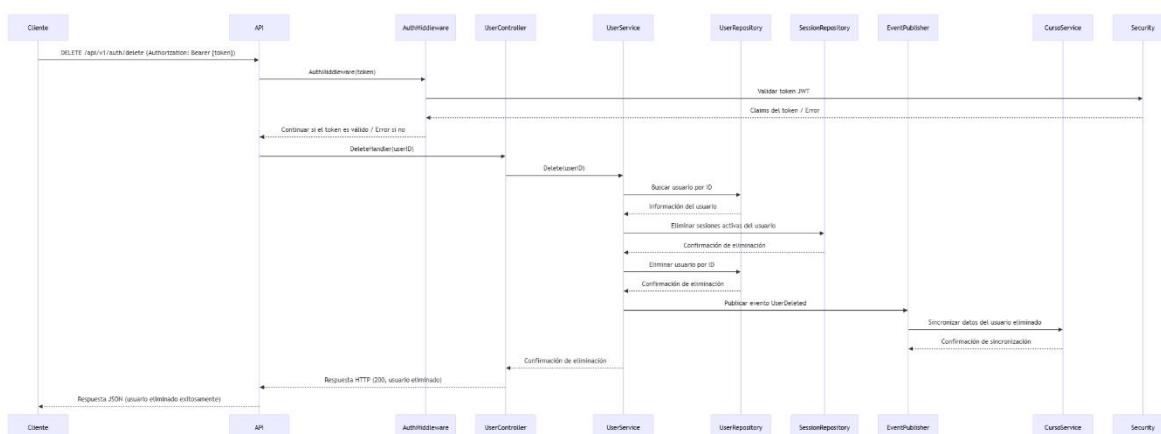
- Obtención de información del usuario a partir del token:



- Cierre de sesión:

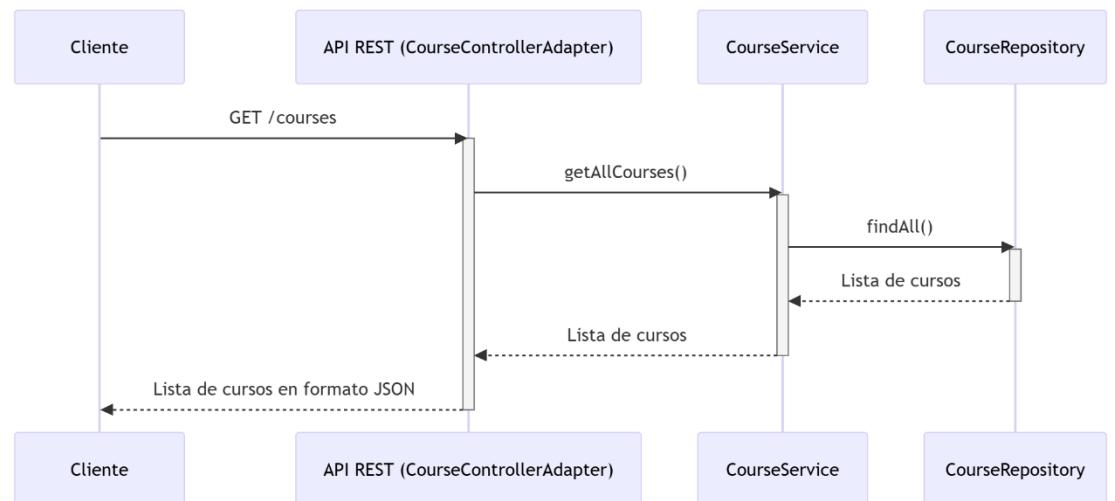


- Eliminación de cuenta de usuario, con propagación de eventos para sincronizar datos en el módulo de cursos:

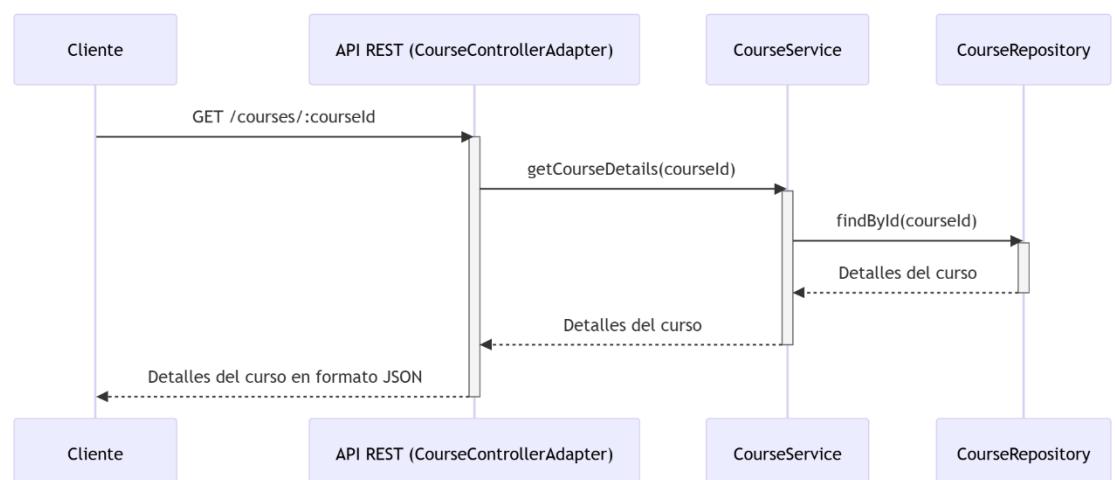


### Aplicación en Java - Gestión de cursos:

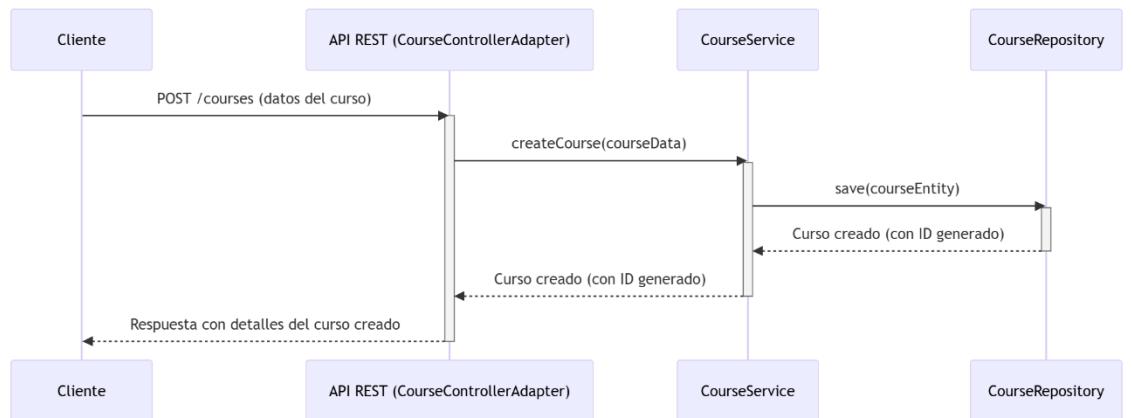
- Consulta de todos los cursos disponibles:



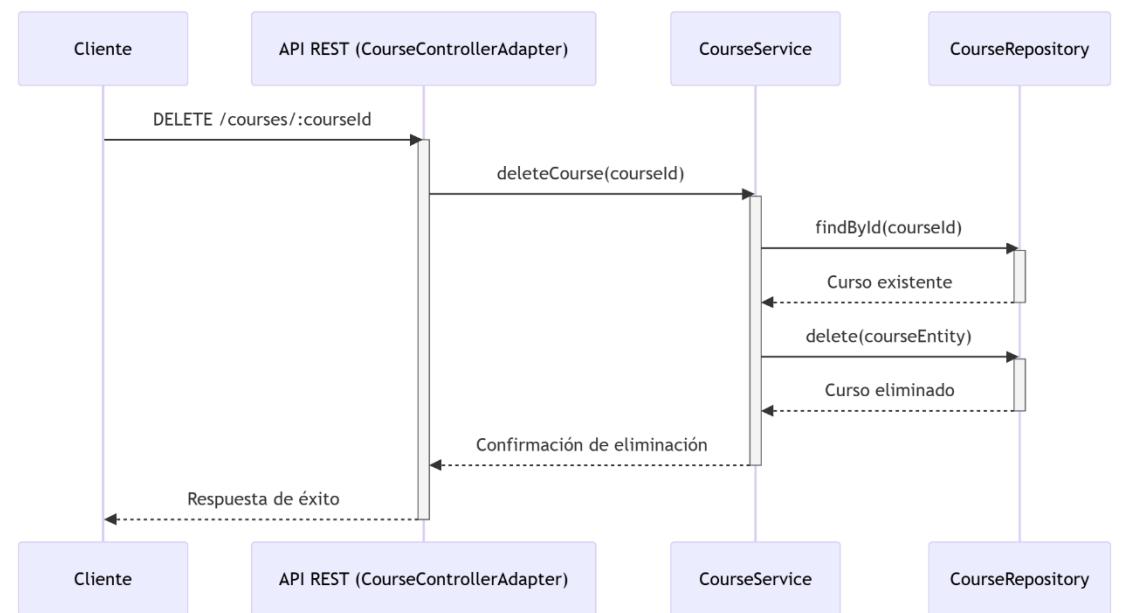
- Acceso a detalles de un curso concreto:



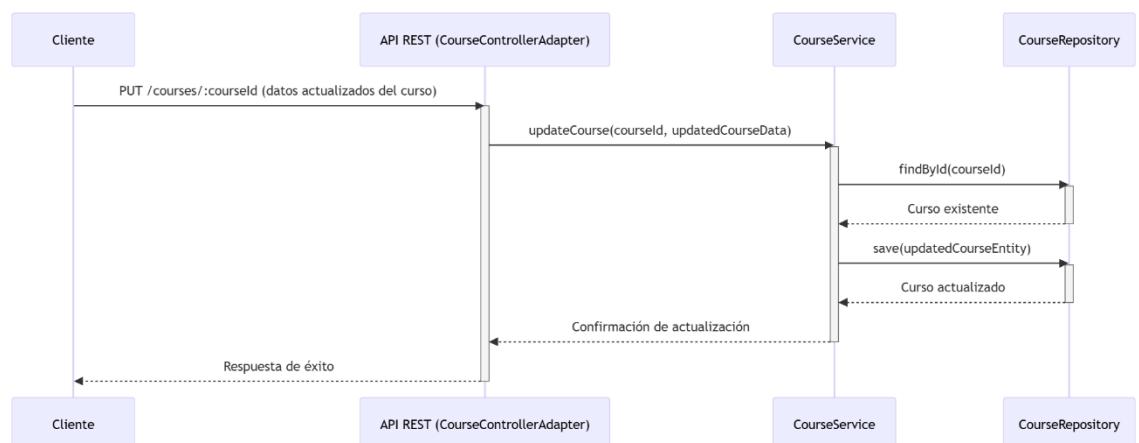
- Creación de un curso:



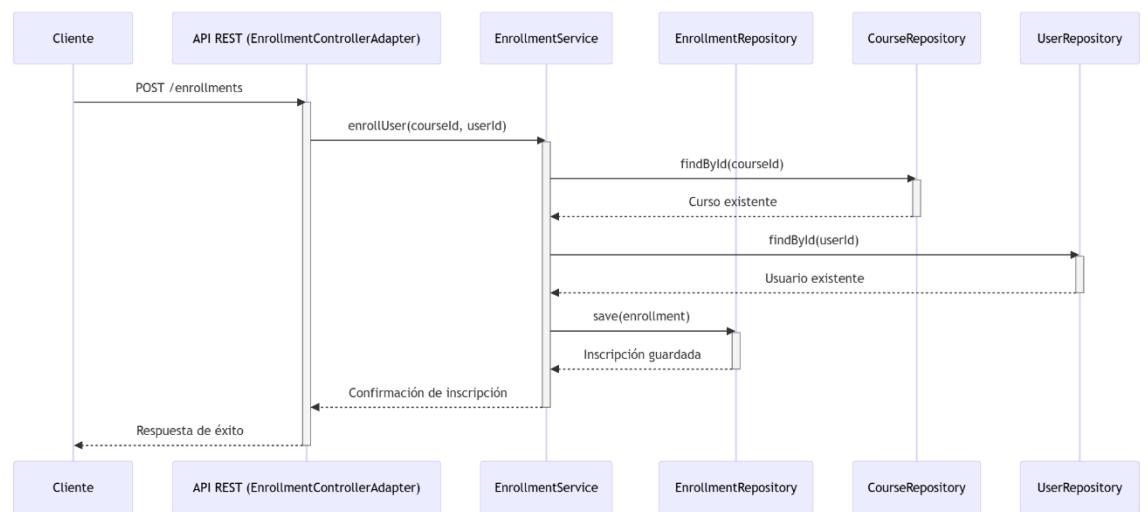
- Actualización de un curso:



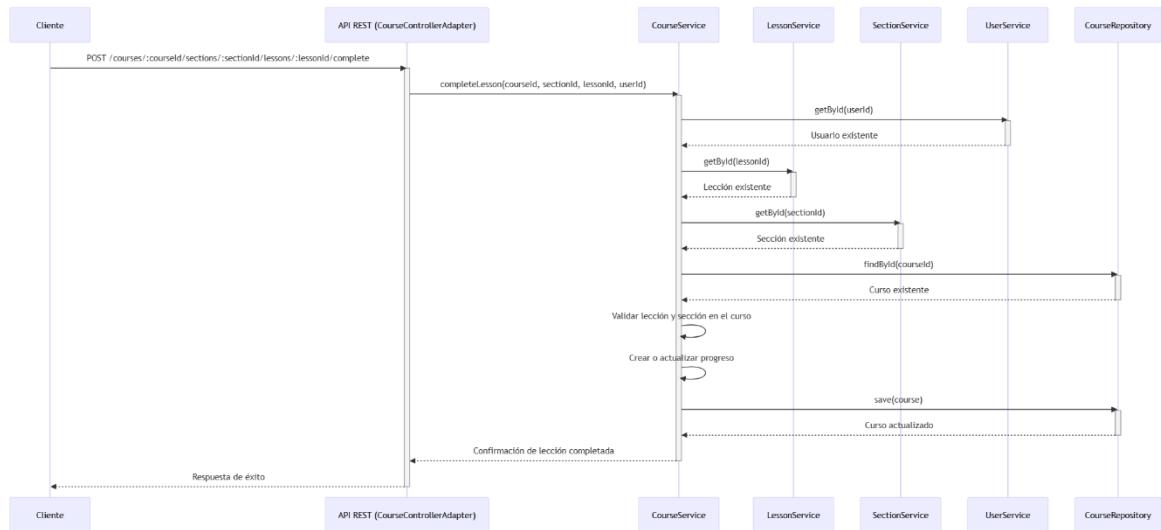
- Eliminación de un curso:



- Inscripción en un curso:



- Completar una lección:

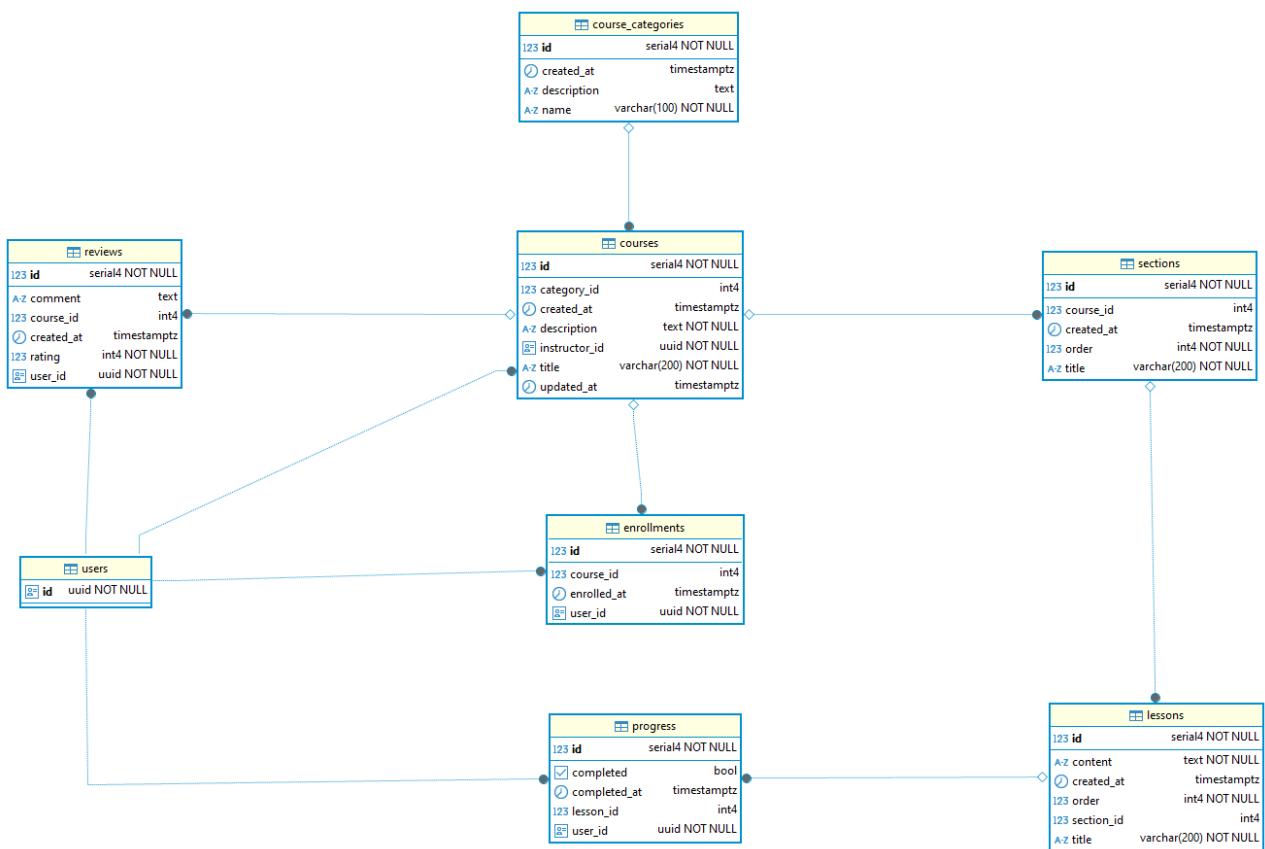


### 4.2.3. Bases de datos – Diagrama entidad / relación

Aplicación en Go - Gestión de usuarios y autenticación:



Aplicación en Java - Gestión de cursos:



#### 4.2.4. Enfoque API First vs Code First

En el presente proyecto se ha optado por un enfoque Code First frente a API First principalmente por dos razones fundamentales: la sencillez de implementación y la necesidad de acelerar el desarrollo dada la limitación temporal. Con Code First, el diseño de la lógica de negocio y de las entidades se realiza directamente sobre el modelo de datos, lo que permite generar automáticamente los esquemas y las rutas de acceso sin necesidad de dedicar tiempo extra a la definición previa de contratos de servicio; de este modo, se reducen las tareas de configuración manual y se agiliza el despliegue inicial de la aplicación.

Además, la prioridad de entregar una versión funcional en plazos ajustados refuerza la idoneidad de Code First, pues minimiza los pasos intermedios y facilita iteraciones rápidas sobre el código. Este procedimiento es especialmente ventajoso cuando los requerimientos no están completamente fijados desde el inicio o pueden sufrir cambios durante la fase de desarrollo, ya que permite adaptar el modelo y la lógica con un impacto mínimo en la infraestructura existente.

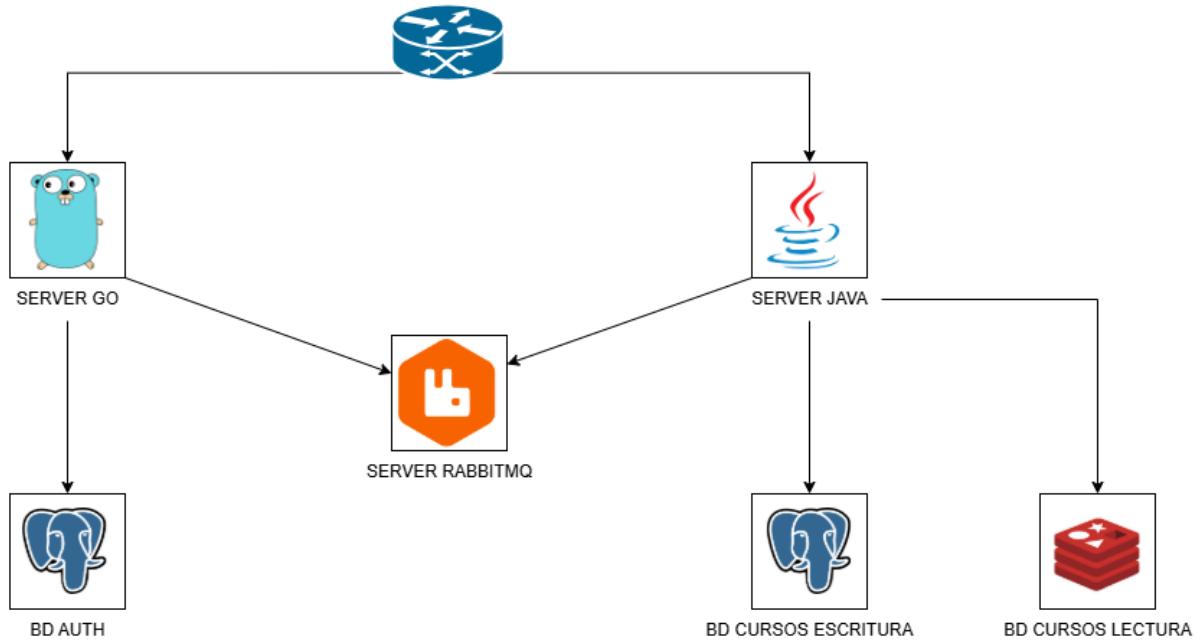
No obstante, es importante destacar que una posible ampliación del proyecto consiste en migrar a un enfoque API First, en el que los contratos de servicio (esquemas OpenAPI) se diseñan y validan de forma previa al desarrollo del código. Esta aproximación aporta mayor estandarización, mejora la interoperabilidad y facilita la colaboración con equipos externos o clientes que consuman los servicios, garantizando además una documentación más precisa y mantenible. Dicha migración y sus implicaciones arquitectónicas se abordarán con detalle en el apartado correspondiente del presente trabajo.



## 4.3. Infraestructura

En este bloque se describe la arquitectura física y lógica que soporta el sistema GoLearnix, incluyendo el esquema de red y el flujo de llamadas entre servidores durante las peticiones.

### 4.3.1. Esquema de red



### 4.3.2. Flujo de llamadas entre servidores durante las peticiones

En el funcionamiento habitual, cada servidor interactúa con su base de datos correspondiente para procesar las peticiones. De forma general, la secuencia típica sigue el patrón:

---

**Cliente → Servidor correspondiente → Base de datos correspondiente.**

---

Sin embargo, debido al uso del patrón **CQRS** (Command Query Responsibility Segregation) en el módulo de gestión de cursos (Java), esta lógica se refina. Para operaciones de lectura (GET), el sistema accede directamente a la base de datos de lectura, implementada con Redis, optimizando la velocidad y eficiencia en la obtención de datos. En cambio, para operaciones de escritura (POST, PUT, DELETE), la petición se gestiona primero en la base de datos principal (PostgreSQL), y posteriormente se actualiza de forma consistente la base de datos de lectura, garantizando así la integridad de los datos entre ambos almacenes.

Una excepción relevante a este flujo general se presenta en la operación de **eliminación de usuario**.

En este caso:

El usuario realiza una petición DELETE al servidor de autenticación (Go).

Este servidor elimina los datos del usuario en su propia base de datos y, si la operación es exitosa, publica un evento en RabbitMQ.

1. El servidor de gestión de cursos (Java) consume este evento, y a partir de él:
2. Elimina al usuario en su base de datos relacional (PostgreSQL).
3. Sincroniza también esta eliminación en su base de datos de lectura (Redis).

Este enfoque basado en eventos permite mantener la coherencia entre servicios desacoplados, al tiempo que se preserva un bajo acoplamiento y una alta escalabilidad del sistema. Además, refuerza la resiliencia ante fallos al distribuir las responsabilidades entre distintos servicios especializados.

#### 4.3.3. Docker

Para facilitar el despliegue y gestión de todos los componentes de la plataforma GoLearnix, se ha utilizado **Docker Compose**. A continuación, se detalla el procedimiento de arranque y parada de la infraestructura, así como la descripción de cada uno de los cuatro servicios definidos en el fichero [docker-compose.yml](#).

Para levantar toda la infraestructura se usa el comando:

`docker compose -p golearnix up -d`

Este comando lee el fichero `docker-compose.yml`, crea la red `golearnix-network`, los volúmenes necesarios y levanta en segundo plano (-d) los cuatro servicios definidos bajo el proyecto golearnix.

Si se desea detener y eliminar los volúmenes se usa el comando:

`docker compose -p golearnix down --volumes`

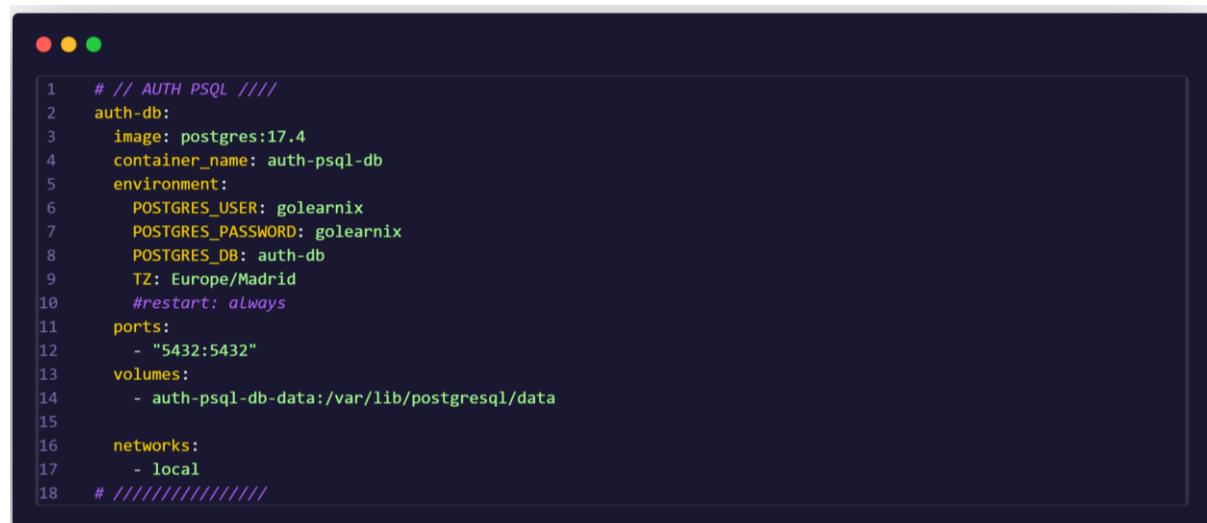


Los servicios se han organizado en dos ámbitos diferenciados: por un lado, la **infraestructura de autenticación**, que agrupa los componentes encargados de validar y gestionar credenciales; y, por otro, la **infraestructura de gestión de cursos**, compuesta por las bases de datos y sistemas de cache y mensajería necesarios para el manejo eficiente de los contenidos formativos.

#### 4.3.3.1. Servicio auth-db (PostgreSQL para autenticación)

El servicio **auth-db** utiliza la imagen oficial *postgres:17.4* y se despliega en el contenedor nombrado *auth-psql-db*. Su propósito es albergar la base de datos de usuarios y credenciales; para ello, se configuran las variables de entorno **POSTGRES\_USER=golearnix**, **POSTGRES\_PASSWORD=golearnix**, **POSTGRES\_DB=auth-db** y **TZ=Europe/Madrid**. Expone el puerto **5432** al host, mapeado a la misma interfaz interna, de modo que las aplicaciones de autenticación puedan conectarse sin necesidad de configuración de red adicional.

Para garantizar la persistencia de datos, monta el volumen *auth-psql-db-data* en */var/lib/postgresql/data*. Finalmente, queda conectado a la red externa *golearnix-network*, lo que permite la comunicación segura y aislada con el resto de los microservicios.



```

1 # // AUTH PSQL /////
2 auth-db:
3   image: postgres:17.4
4   container_name: auth-psql-db
5   environment:
6     POSTGRES_USER: golearnix
7     POSTGRES_PASSWORD: golearnix
8     POSTGRES_DB: auth-db
9     TZ: Europe/Madrid
10    #restart: always
11  ports:
12    - "5432:5432"
13  volumes:
14    - auth-psql-db-data:/var/lib/postgresql/data
15
16  networks:
17    - local
18 # ///////////////////

```

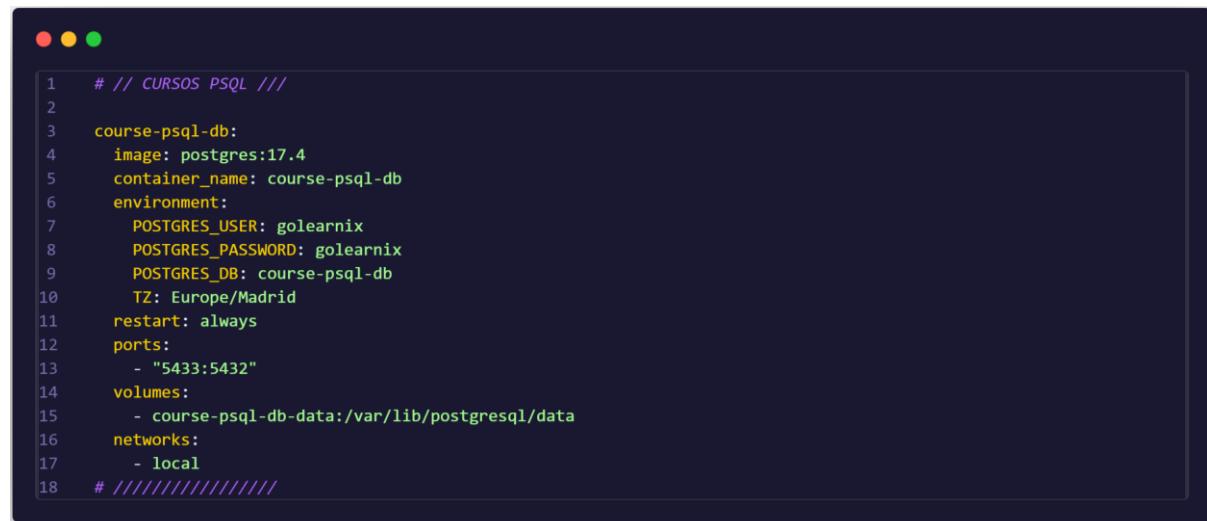
#### 4.3.3.2. Servicio course-psql-db (PostgreSQL para gestión de cursos)

El servicio **course-psql-db** emplea la misma versión de PostgreSQL (*postgres:17.4*) y se instancia en el contenedor *course-psql-db*. Sus variables de entorno, **POSTGRES\_USER=golearnix**, **POSTGRES\_PASSWORD=golearnix**, **POSTGRES\_DB=course-psql-db** y **TZ=Europe/Madrid**, definen el entorno de ejecución y la base de datos asociada. Se configura reinicio automático (*always*) y se



expone el puerto **5433** en el host, redirigiéndolo al 5432 interno, con el fin de evitar colisiones en entornos locales donde ya exista otro servidor PostgreSQL.

Para la persistencia, dispone del volumen `course-psql-db-data` montado en `/var/lib/postgresql/data`. Este servicio se integra igualmente en la red compartida `golearnix-network`, permitiendo a las API de cursos y otros microservicios acceder a la información curricular.



```

1 # // CURSOS PSQL /**
2
3 course-psql-db:
4   image: postgres:17.4
5   container_name: course-psql-db
6   environment:
7     POSTGRES_USER: golearnix
8     POSTGRES_PASSWORD: golearnix
9     POSTGRES_DB: course-psql-db
10    TZ: Europe/Madrid
11   restart: always
12   ports:
13     - "5433:5432"
14   volumes:
15     - course-psql-db-data:/var/lib/postgresql/data
16   networks:
17     - local
18 # ///////////////////

```

#### 4.3.3.3. Servicio redis (Redis Stack para caché y colas ligeras)

El contenedor `course-redis-db`, basado en `redis/redis-stack:7.4.0-v3`, ofrece un sistema de caché y colas en memoria de alto rendimiento. Con política de *reinicio always* y *zona horaria Europe/Madrid*, expone el puerto **6379** para conexiones de cliente.

Se montan tres volúmenes:

- `course-redis-db-data` en `/data`, que opcionalmente persiste el estado de Redis;
- `./resources/databases/course-redis-db/config/redis.conf` en `/usr/local/etc/redis/redis.conf`, donde se define la configuración principal del servidor;
- `./resources/databases/course-redis-db/config/acl.conf` en `/etc/redis/acl.conf`, para el control de acceso de usuarios.

Finalmente, este servicio también forma parte de `golearnix-network`.



```

1 # // CURSOS REDIS //
2
3 redis:
4   image: redis/redis-stack:7.4.0-v3
5   container_name: course-redis-db
6   restart: always
7   environment:
8     TZ: Europe/Madrid
9   ports:
10    - "6379:6379"
11   volumes:
12    - course-redis-db-data:/data
13    - ./resources/databases/course-redis-db/config/redis.conf:/usr/local/etc/redis/redis.conf
14    - ./resources/databases/course-redis-db/config/acl.conf:/etc/redis/acl.conf
15   command: [ "redis-stack-server", "/usr/local/etc/redis/redis.conf" ]
16   networks:
17    - local
18 # /////////////////

```

#### 4.3.3.4. Servicio rabbit (RabbitMQ con consola de gestión)

El servicio **rabbit**, basado en *rabbitmq:3-management*, se configura con el nombre de contenedor rabbit y el hostname interno rabbit. Se establece política de reinicio permanente (*restart: always*) para garantizar disponibilidad continua. Las credenciales por defecto se fijan en **RABBITMQ\_DEFAULT\_USER=golearnix** y **RABBITMQ\_DEFAULT\_PASS=golearnix**.

Este broker AMQP ofrece tanto el puerto estándar **5672** para mensajería como el **15672** para el acceso web a su interfaz de administración. Para la configuración inicial, se montan en modo lectura los ficheros **definitions.json** y **rabbitmq.conf** desde el directorio *./resources/rabbitmq/*, además de un volumen de datos *rabbitmq-data* en */var/lib/rabbitmq* que almacena colas, exchanges y metadatos. Al igual que el resto, forma parte de **golearnix-network**.

```

1 # // RABBITMQ ////
2 rabbit:
3   image: rabbitmq:3-management
4   container_name: rabbit
5   hostname: rabbit
6   restart: always
7   environment:
8     RABBITMQ_DEFAULT_USER: golearnix
9     RABBITMQ_DEFAULT_PASS: golearnix
10  ports:
11    - "5672:5672"
12    - "15672:15672"
13  volumes:
14    - ./resources/rabbitmq/definitions.json:/etc/rabbitmq/definitions.json:ro
15    - ./resources/rabbitmq/rabbitmq.conf:/etc/rabbitmq/rabbitmq.conf:ro
16    - rabbitmq-data:/var/lib/rabbitmq
17  networks:
18    - local
19 # ///////////////

```



**El fichero definitions.json define la configuración inicial de la topología de RabbitMQ:** crea el *vhost raíz* (/), registra el usuario golearnix con permisos completos sobre ese vhost (configurar, escribir y leer), y establece dos colas durables (*user.deleted.queue* y su *dead-letter queue user.deleted.dlq*). Además, declara *dos exchanges de tipo topic* (*golearnix.events* y *golearnix.events.dlx*) y vincula cada exchange con su correspondiente cola mediante claves de ruteo (*user.deleted* y *user.deleted.dlq*), garantizando el procesamiento y reencolado automático de mensajes rechazados.

```

1  {
2    "vhosts": [
3      {
4        "name": "/"
5      }
6    ],
7    "users": [
8      {
9        "name": "golearnix",
10       "password": "golearnix",
11       "tags": "administrator"
12     }
13   ],
14   "permissions": [
15     {
16       "user": "golearnix",
17       "vhost": "/",
18       "configure": ".*",
19       "write": ".*",
20       "read": ".*"
21     }
22   ],
23   "queues": [
24     {
25       "name": "user.deleted.queue",
26       "vhost": "/",
27       "durable": true,
28       "arguments": {
29         "x-dead-letter-exchange": "golearnix.events.dlx",
30         "x-dead-letter-routing-key": "user.deleted.dlq"
31       }
32     },
33     {
34       "name": "user.deleted.dlq",
35       "vhost": "/",
36       "durable": true
37     }
38   ],
39   "exchanges": [
40     {
41       "name": "golearnix.events",
42       "vhost": "/",
43       "type": "topic",
44       "durable": true
45     },
46     {
47       "name": "golearnix.events.dlx",
48       "vhost": "/",
49       "type": "topic",
50       "durable": true
51     }
52   ],
53   "bindings": [
54     {
55       "source": "golearnix.events",
56       "vhost": "/",
57       "destination": "user.deleted.queue",
58       "destination_type": "queue",
59       "routing_key": "user.deleted"
60     },
61     {
62       "source": "golearnix.events.dlx",
63       "vhost": "/",
64       "destination": "user.deleted.dlq",
65       "destination_type": "queue",
66       "routing_key": "user.deleted.dlq"
67     }
68   ]
69 ]
70 }

```



El fichero **rabbitmq.conf** define los parámetros de arranque y gestión del broker. En concreto, incluye la directiva:

```
● ● ●
1 management.load_definitions = /etc/rabbitmq/definitions.json
2
```

Que indica al servicio de administración que cargue automáticamente las definiciones de *exchanges*, colas, usuarios y permisos desde el fichero *definitions.json* cada vez que el contenedor se inicia. De este modo, se asegura la reproducibilidad de la topología sin intervención manual.

#### 4.3.3.5. Red y volúmenes compartidos

La totalidad de los contenedores se conecta a una red externa denominada golearnix-network. De este modo, se garantiza el aislamiento del tráfico y la interoperabilidad entre proyectos.

Para la persistencia de datos críticos se declaran cuatro volúmenes. Cada volumen asegura la durabilidad de las bases de datos y colas, evitando la pérdida de información ante reinicios o recreaciones de contenedores.

```
● ● ●
1 networks:
2   local:
3     external: true
4     name: golearnix-network
5
6 volumes:
7   course-redis-db-data:
8   course-psql-db-data:
9   auth-psql-db-data:
10  rabbitmq-data:
```



## 5. IMPLEMENTACIÓN DEL PROYECTO

En este apartado se describen en detalle las decisiones técnicas adoptadas durante el desarrollo del proyecto. Se analizan las tecnologías seleccionadas, la arquitectura del sistema y los criterios que han guiado la implementación. Además, se presenta el funcionamiento del código desarrollado, explicando su estructura, los principales módulos que lo componen y el proceso seguido para su construcción. El objetivo es proporcionar una visión clara y fundamentada sobre cómo se ha llevado a cabo la implementación práctica del sistema propuesto.

El código fuente completo del proyecto, así como su documentación técnica, se encuentra disponible para su consulta en el siguiente enlace:

<https://github.com/javferTec/GoLearnix>



### 5.1. Aplicación con Go para autenticación y gestión de usuarios

Este microservicio ha sido implementado en Go utilizando el framework Fiber. Su responsabilidad principal es la gestión de usuarios y todo lo relativo a los procesos de autenticación y autorización dentro de la arquitectura de la aplicación.

#### 5.1.1. ¿Por qué usar Go?

Go (o Golang) se ha seleccionado como lenguaje de programación por las siguientes razones de las que ya se ha comentado en más profundidad en el marco conceptual:

- **Alto rendimiento y eficiencia**

Go compila a binarios nativos que aprovechan al máximo los recursos del sistema, logrando tiempos de arranque rápidos y un footprint de memoria reducido.

- **Concurrencia sencilla y robusta**

Su modelo de goroutines y canales simplifica la escritura de código concurrente, permitiendo gestionar múltiples tareas asíncronas de forma segura.

- **Ecosistema y madurez de librerías**

Un amplio conjunto de paquetes oficiales y de terceros, junto con el framework Fiber, facilita el desarrollo ágil de microservicios RESTful.

Fiber destaca frente a otros frameworks de Go al estar construido sobre fasthttp, lo que le permite ofrecer un rendimiento excepcional con tasas de solicitudes por segundo muy elevadas y latencias mínimas; a esto se suma su API inspirada en Express.js, que facilita la adopción por parte de desarrolladores con experiencia en JavaScript al mantener una sintaxis clara y coherente para enrutamiento, middleware y manejo de errores. Su sistema de middleware es modular y ligero, de modo que se pueden incorporar o descartar componentes según las necesidades sin sacrificar velocidad, y permite agrupar rutas o montar subaplicaciones para estructurar proyectos de gran envergadura con una sobrecarga mínima. Además, Fiber incluye de serie soporte optimizado para WebSocket, renderizado de plantillas (HTML, Pug, Handlebars, etc.) y compresión HTTP, evitando la dependencia de librerías externas y garantizando que todas las funcionalidades operen bajo el mismo motor de alto rendimiento. Finalmente, su excelente documentación y la comunidad activa facilitan la resolución de dudas, la consulta de ejemplos completos y la contribución al proyecto, lo que refuerza su adopción en entornos profesionales.

### 5.1.2. ¿Por qué usar UUID como tipo de dato para el identificador de los usuarios?

Los UUID (Universally Unique Identifier) generan identificadores de 128 bits con una probabilidad prácticamente nula de colisión, incluso cuando múltiples instancias del microservicio crean recursos de forma concurrente y en diferentes nodos. Esto evita la necesidad de coordinar un contador centralizado y elimina puntos únicos de fallo.

Al usar este tipo de dato como clave primaria, los identificadores pueden generarse en la capa de aplicación antes de persistir el recurso. Esto facilita la integración con sistemas externos (colas de mensajes, APIs de terceros) sin depender de la respuesta de la base de datos para conocer el ID del usuario.

A diferencia de los IDs secuenciales, que pueden permitir inferir el número total de usuarios o recursos existentes, los UUID son no lineales y no predecibles. Esto dificulta ataques de enumeración por parte de un usuario malintencionado.



Además, pueden representarse en diferentes formatos (texto, binario, base64) y son compatibles con la mayoría de las bases de datos relacionales y no relacionales. Su adopción estandarizada (RFC 4122) garantiza interoperabilidad entre sistemas heterogéneos.

### 5.1.3. ¿Cómo generar el contrato Swagger a partir de los controladores?

Se debe instalar la herramienta de generación de documentación Swagger swag mediante el comando

```
go install github.com/swaggo/swag/cmd/swag@latest
```

Asegurándose de que el directorio `$GOPATH/bin` figure en la variable PATH para poder invocar swag globalmente. A continuación, se incluyen en cada controlador las anotaciones compatibles con Swaggo (`@Summary`, `@Description`, `@Tags`, `@Param`, `@Success`, `@Failure`, `@Router` y, cuando procede, `@Security BearerAuth`), así como la definición global de seguridad en el paquete principal:

```

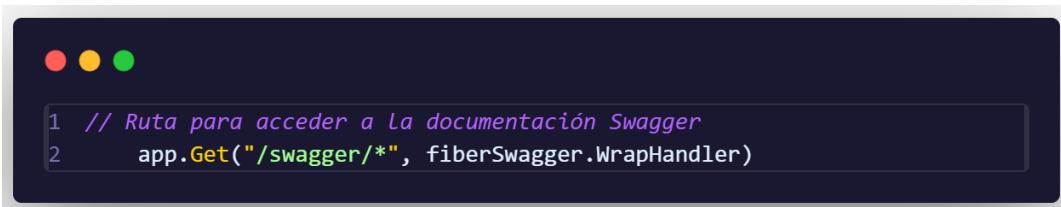
1 // MeHandler obtiene información del usuario actual.
2 // @Summary Obtener información del usuario actual
3 // @Description Devuelve los datos del usuario autenticado.
4 // @Tags User
5 // @Security BearerAuth
6 // @Accept json
7 // @Produce json
8 // @Success 200 {object} response.Response{data=map[string]interface{}} "Información del usuario"
9 // @Failure 401 {object} response.Response "No autorizado"
10 // @Failure 500 {object} response.Response "Error interno del servidor"
11 // @Router /api/v1/auth/me [get]
12 func (uc *UserController) MeHandler(c *fiber.Ctx) error {
13     userID := c.Locals("userID").(string)
14     user, err := uc.UserService.GetMe(userID)
15     if err != nil {
16         log.Printf("Error al obtener la información del usuario: %v", err)
17         return response.Error(c, fiber.StatusInternalServerError, "No se pudo obtener el usuario")
18     }
19     return response.Success(c, fiber.StatusOK, "Información del usuario", fiber.Map{
20         "user": user,
21     })
22 }
```

Para generar el contrato OpenAPI, desde la raíz del módulo (donde reside go.mod) se ejecuta:

```
swag init --output ./docs --generalInfo presentation/api/swagger.go --parseDependency
```



De este modo, se vuelcan en `./docs` los archivos `swagger.json` y `swagger.yaml`, incorporando tanto las anotaciones de la aplicación como las de sus dependencias. En el arranque de la aplicación Fiber se importa el paquete generado con un blank import (`_ "golearnix-auth/docs"`) y se expone la ruta de la interfaz Swagger UI mediante:



```

1 // Ruta para acceder a La documentación Swagger
2 app.Get("/swagger/*", fiberSwagger.WrapHandler)

```

Así, al acceder a <http://localhost:2003/swagger/index.html>, el contrato de la API quedará disponible de forma interactiva. Cada modificación en los controladores debe ir acompañada de una nueva ejecución de `swag init` para mantener la documentación sincronizada con el código fuente.

**NOTA:** *2003 es el puerto en el que se ejecuta el servidor. Se establece a través de:*



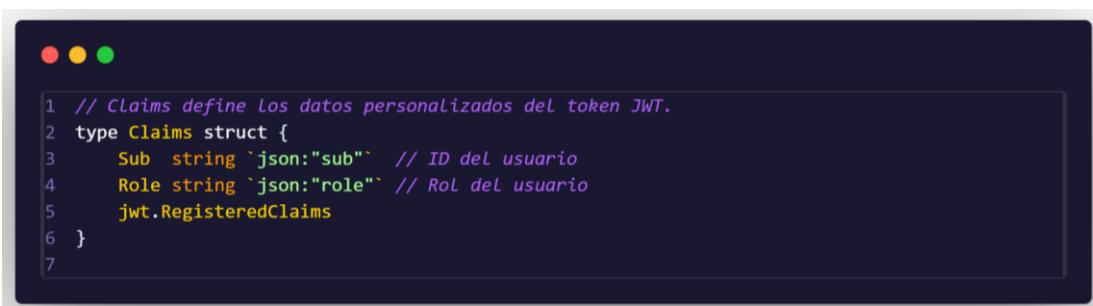
```

1 // Iniciar servidor
2 log.Fatal(app.Listen(":2003"))

```

#### 5.1.4. ¿Cómo generar el Json Web Token y validararlo?

Se ha definido en el paquete security una estructura de `Claims` que extiende `jwt.RegisteredClaims`, incorporando los campos `sub` (identificador del usuario) y `role` (rol asignado), tras los cuales se declara la variable `jwtSecret` que, en la función `init()`, se inicializa decodificando en bytes la cadena hexadecimal obtenida de la variable de entorno `JWT_SECRET`.



```

1 // Claims define Los datos personalizados del token JWT.
2 type Claims struct {
3     Sub string `json:"sub"` // ID del usuario
4     Role string `json:"role"` // Rol del usuario
5     jwt.RegisteredClaims
6 }
7

```



```

1 // jwtSecret contiene los bytes decodificados desde el hex en JWT_SECRET
2 var jwtSecret []byte
3
4 func init() {
5     if err := godotenv.Load(); err != nil {
6         log.Println("⚠️ Advertencia: no se pudo cargar .env en security.init()")
7     }
8
9     hexKey := os.Getenv("JWT_SECRET")
10
11    if hexKey == "" {
12        log.Fatal("JWT_SECRET no configurado")
13    }
14
15    keyBytes, err := hex.DecodeString(hexKey)
16
17    if err != nil {
18        log.Fatalf("error al decodificar el JWT_SECRET desde hex: %v", err)
19    }
20
21    jwtSecret = keyBytes
22 }

```

La generación de un token se lleva a cabo mediante la función GenerateToken, que recibe el UUID del usuario, su rol, un identificador único de token (JTI) y la duración deseada; a partir de estos parámetros, construye un objeto Claims con su fecha de emisión (IssuedAt) y de expiración (ExpiresAt), firma el JWT empleando el algoritmo HS256 y la clave secreta previamente decodificada, y retorna la cadena resultante.

```

1 // GenerateToken genera un JWT para un usuario con duración específica y JTI.
2 func GenerateToken(userID uuid.UUID, role string, jwtID string, duration time.Duration) (string, error) {
3     // Crea los claims del token
4     claims := Claims{
5         Sub: userID.String(),
6         Role: role,
7         RegisteredClaims: jwt.RegisteredClaims{
8             ID:        jwtID,
9             ExpiresAt: jwt.NewNumericDate(time.Now().Add(duration)),
10            IssuedAt: jwt.NewNumericDate(time.Now()),
11        },
12    }
13
14    token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
15
16    // Firmar el token
17    signedToken, err := token.SignedString(jwtSecret)
18    if err != nil {
19        return "", fmt.Errorf("error al firmar el token: %v", err)
20    }
21
22    // Retornar el token firmado
23    return signedToken, nil
24 }

```

La validación del token se efectúa en ValidateToken, que invoca jwt.ParseWithClaims para reconstruir el objeto Claims contenidos en el JWT y comprueba tanto la integridad de la firma como su vigencia;



si el parsing resulta exitoso y el token se considera válido, devuelve los claims deserializados, y en caso contrario propaga el error correspondiente, diferenciando entre firma inválida y otros fallos de parsing o expiración.



```

1 // Valida el JWT y devuelve los claims si es válido.
2 func ValidateToken(tokenString string) (*Claims, error) {
3     // Parsea el token con los claims definidos
4     token, err := jwt.ParseWithClaims(tokenString, &Claims{}, func(token *jwt.Token) (interface{}, error) {
5         return jwtSecret, nil
6     })
7
8     if err != nil {
9         return nil, err
10    }
11
12    claims, ok := token.Claims.(*Claims)
13    if !ok || !token.Valid {
14        return nil, jwt.ErrSignatureInvalid
15    }
16
17    return claims, nil
18 }
```

Este enfoque permite gestionar de manera centralizada y segura la autenticación basada en JWT, garantizando la confidencialidad de la clave secreta, la inclusión de metadatos del usuario en el propio token y un mecanismo fiable de verificación de validez temporal y de integridad criptográfica.

### 5.1.5. ¿Cómo conectar con la base de datos con GORM?

Se ha optado por GORM como ORM en Go debido a su capacidad para mapear estructuras de Go a tablas relacionales de forma transparente, manejar operaciones CRUD sin SQL explícito y facilitar migraciones automáticas. GORM abstrae la interacción con la base de datos mediante una API fluida basada en métodos encadenados, soporte para transacciones, relaciones entre modelos y hooks para personalizar la lógica antes o después de cada operación.

La inicialización de la conexión se realiza en el paquete `database` a través de la función `Connect`, que extrae la cadena de conexión (`DATABASE_URL`) del entorno, abre un enlace con PostgreSQL mediante `gorm.Open(postgres.Open(dsn), &gorm.Config{})` y, tras comprobar errores, invoca `db.AutoMigrate(&models2.User{}, &models2.Session{})` para sincronizar los esquemas de las entidades `User` y `Session` con la base de datos. Finalmente, la instancia resultante se asigna a la variable global `DB`, de modo que pueda reutilizarse en todo el microservicio para realizar consultas, inserciones y demás operaciones sobre las entidades persistentes.



```

1 // Connect inicializa la conexión a la base de datos y realiza las migraciones.
2 func Connect() error {
3     // Obtener la URL de conexión a la base de datos desde el entorno
4     dsn := os.Getenv("DATABASE_URL")
5     if dsn == "" {
6         log.Fatal("DATABASE_URL no está configurada en el entorno")
7     }
8
9     // Conectar con la base de datos
10    db, err := gorm.Open(postgres.Open(dsn), &gorm.Config{})
11    if err != nil {
12        return err
13    }
14
15    if err := db.AutoMigrate(
16        &models2.User{},
17        &models2.Session{},
18    ); err != nil {
19        return err
20    }
21
22    DB = db // Registrar la conexión a la base de datos
23
24    return nil
25 }
26

```

### 5.1.6. ¿Cómo conectar con él gestor de eventos RabbitMQ?

Se utiliza la librería oficial [github.com/rabbitmq/amqp091-go](https://github.com/rabbitmq/amqp091-go), que implementa el protocolo AMQP 0-9-1 y proporciona una abstracción para gestionar conexiones, canales y publicaciones de mensajes. Para establecer la conexión, se invoca `amqp091.Dial(amqpURL)`, pasando la URL de RabbitMQ (incluyendo credenciales, host y puerto), y una vez obtenida la `*amqp091.Connection` se crea un canal de comunicación con `conn.Channel()`. Ambos objetos se encapsulan en la estructura `EventPublisher`, de modo que permanezcan disponibles para operaciones de publicación sobre `publisher.channel`. Al finalizar el ciclo de vida de la aplicación, `publisher.Close()` se encarga de cerrar primero el canal y luego la conexión, garantizando la liberación ordenada de recursos y evitando posibles fugas de sockets.

```

1 // EventPublisher es la estructura que maneja la conexión y el canal de RabbitMQ
2 type EventPublisher struct {
3     conn    *amqp091.Connection
4     channel *amqp091.Channel
5 }

```



```

1 // NewEventPublisher establece la conexión y canal con RabbitMQ,
2 func NewEventPublisher(amqpURL string) (*EventPublisher, error) {
3     conn, err := amqp091.Dial(amqpURL)
4     if err != nil {
5         return nil, fmt.Errorf("error conectando a RabbitMQ: %w", err)
6     }
7
8     ch, err := conn.Channel()
9     if err != nil {
10        return nil, fmt.Errorf("error creando canal de RabbitMQ: %w", err)
11    }
12
13    return &EventPublisher{conn: conn, channel: ch}, nil
14 }
15
16 // Close cierra la conexión y el canal
17 func (p *EventPublisher) Close() {
18     if err := p.channel.Close(); err != nil {
19         fmt.Printf("Error cerrando canal: %v\n", err)
20     }
21
22     if err := p.conn.Close(); err != nil {
23         fmt.Printf("Error cerrando conexión: %v\n", err)
24     }
25 }

```

### 5.1.7. Arquitectura de 3 capas

La solución adopta una arquitectura de tres capas que separa de forma rigurosa las responsabilidades de presentación, lógica de negocio y acceso a datos, lo que favorece la mantenibilidad, la escalabilidad y la posibilidad de evolución independiente de cada componente.

En la capa de presentación, ubicada bajo el directorio **presentation**, se encuentran los controladores HTTP, las rutas y los middleware de autenticación. Los ficheros `auth_routes.go` y `user_routes.go` definen de manera declarativa los endpoints expuestos, mientras que `auth_controller.go` y `user_controller.go` implementan la lógica de traducción entre las peticiones REST y los servicios de dominio. El middleware `auth_middleware.go` se encarga de interceptar las solicitudes protegidas, extraer y validar el JWT, y propagar la identidad del usuario al contexto de ejecución, de modo que los controladores puedan operar sobre un modelo de autorización homogéneo.

La capa de dominio, en **domain**, centraliza las reglas de negocio y los modelos de datos. Dentro de **models** se definen las entidades `User`, `Session` y `UserRole`, reflejo directo de las tablas de la base de datos, mientras que en **dto/request** y **dto/events** se declaran los objetos de transferencia que permiten desacoplar la capa de presentación de los detalles internos de las entidades. Los repositorios (**domain/repositories**) exponen interfaces que abstraen las operaciones sobre persistencia, aplicando el principio de inversión de dependencia y los servicios (**domain/services** y su paquete `imp`) implementan la lógica de registro, login, validación de tokens y gestión de sesiones, coordinando



Llamadas a repositorios, generación de eventos y utilidades transversales (respuesta, seguridad, etc.). Estos servicios mantienen la consistencia de las reglas de negocio y generan los eventos de usuario encapsulados en **domain/dto/events** para ser publicados a través del bus de mensajería.

La capa de acceso a datos se reparte entre el paquete **datasource/database**, responsable de la conexión centralizada con PostgreSQL y las migraciones automáticas mediante GORM, y **datasource/repositories**, que ofrece implementaciones concretas de los repositorios definidos en dominio ([GormUserRepository](#), [GormSessionRepository](#)).

De forma complementaria, existe un submódulo de eventos ([events/rabbit/publishers](#)) que encapsula la interacción con RabbitMQ.

Finalmente, el archivo [main.go](#) orquesta la inicialización de todas las capas: carga de variables de entorno, conexión a la base de datos, configuración de repositorios y servicios, instancia de controladores, montaje de rutas y exposición de la documentación Swagger. Gracias a esta organización en tres capas clásicas: presentación, dominio y persistencia, junto con un módulo de eventos desacoplado, el sistema alcanza un alto grado de cohesión interna y un bajo acoplamiento entre componentes, lo que se traduce en facilidad de testing y de evolución futura sin comprometer módulos no relacionados.

```

├── datasource
│   ├── database
│   └── repositories
├── docs
└── domain
    ├── dto
    │   ├── events
    │   └── request
    ├── models
    ├── repositories
    ├── services
    │   └── impl
    └── utils
        ├── email
        ├── response
        └── security
├── events
│   └── rabbit
│       └── publishers
└── presentation
    ├── api
    ├── controllers
    └── middlewares

```



### 5.1.8. Modelos de datos de dominio

A continuación, se describen los tres modelos que representan las entidades principales del dominio de autenticación y gestión de usuarios. Cada modelo se acompaña de su correspondiente representación gráfica.

El modelo **User** encapsula la información esencial de cada usuario registrado en el sistema:

- **ID (uuid.UUID)**: Identificador único de tipo UUID, generado automáticamente por PostgreSQL (`uuid_generate_v4()`) y utilizado como clave primaria.
- **Name (string)**: Nombre completo del usuario, con longitud máxima de 100 caracteres y marcado como obligatorio.
- **Email (string)**: Correo electrónico único, índice único y campo obligatorio de hasta 255 caracteres.
- **PasswordHash (string)**: Hash de la contraseña (campo oculto en la serialización JSON), almacenado de forma segura mediante algoritmos de hashing.
- **Role (UserRole)**: Rol asignado al usuario (admin, instructor o student), definido en un tipo enumerado y con valor por defecto student.
- **CreatedAt (time.Time)**: Marca temporal de creación automática.
- **UpdatedAt (time.Time)**: Marca temporal de última actualización automática.

```

● ● ●
1 type User struct {
2     ID        uuid.UUID `gorm:"type:uuid;default:uuid_generate_v4();primaryKey" json:"id"`
3     Name      string   `gorm:"size:100;not null" json:"name"`
4     Email     string   `gorm:"size:255;unique;not null" json:"email"`
5     PasswordHash string   `gorm:"not null" json:"-"`
6     Role      UserRole `gorm:"type:user_role;default:'student';not null" json:"role"`
7     CreatedAt time.Time `gorm:"autoCreateTime" json:"created_at"`
8     UpdatedAt time.Time `gorm:"autoUpdateTime" json:"updated_at"`
9 }

```

El tipo **UserRole** es un alias de string que define los roles disponibles en la aplicación:

- **Admin**: Usuario con permisos administrativos completos.
- **Instructor**: Usuario con permisos para gestionar contenidos y supervisar alumnos.
- **Student**: Usuario estándar, con acceso restringido a sus propios datos y funcionalidades de aprendizaje.



```

 1 const (
 2     Admin      UserRole = "admin"
 3     Instructor UserRole = "instructor"
 4     Student    UserRole = "student"
 5 )
 6

```

El modelo **Session** representa una sesión activa de usuario y sirve para invalidar tokens JWT según su identificador (JwtID):

- **ID (uuid.UUID)**: Identificador único de la sesión, utilizado como clave primaria.
- **UserID (uuid.UUID)**: Clave foránea que referencia al usuario propietario de la sesión, indexada para consultas eficientes.
- **JwtID (uuid.UUID)**: Identificador único del token (JTI), almacenado para permitir la revocación individual de JWT.
- **IssuedAt (time.Time)**: Fecha y hora de emisión del token.
- **ExpiresAt (time.Time)**: Fecha y hora de expiración del token.

```

 1 type Session struct {
 2     ID      uuid.UUID `gorm:"type:uuid;default:uuid_generate_v4();primaryKey" json:"id"`
 3     UserID  uuid.UUID `gorm:"type:uuid;not null;index:user_id_index" json:"user_id"`
 4     JwtID  uuid.UUID `gorm:"type:uuid;not null;unique" json:"jwt_id"`
 5     IssuedAt time.Time `gorm:"not null" json:"issued_at"`
 6     ExpiresAt time.Time `gorm:"not null" json:"expires_at"`
 7 }

```

Estos modelos, definidos bajo el paquete **domain/models**, forman la base de datos del dominio y se mapean automáticamente a tablas relacionales mediante GORM, garantizando un acceso tipado, coherente y alineado con las reglas de negocio.

### 5.1.9. Exposición de rutas y endpoints

La configuración de las rutas HTTP se organiza en el paquete `presentation/api`, donde se agrupan de forma modular las rutas de autenticación y las del usuario autenticado, aplicando de manera consistente el middleware de seguridad para las operaciones protegidas.



## Inicialización general

En la función `SetupRoutes` ([archivo api/api.go](#)) se inyectan las dependencias principales, instancias de `AuthController`, `UserController` y el repositorio de sesiones, y se delega la creación de cada conjunto de rutas a sus respectivos métodos especializados:

```
1 func SetupRoutes(app *fiber.App, authController *controllers2.AuthController, userController *controllers2.UserController, sessionRepo repositories.SessionRepository) {
2     // Configurar rutas de autenticación
3     SetupAuthRoutes(app, authController, sessionRepo)
4
5     // Configurar rutas del usuario
6     SetupUserRoutes(app, userController, sessionRepo)
7 }
```

## Rutas de autenticación

Definidas en [api/auth\\_routes.go](#), las rutas de autenticación se agrupan bajo el prefijo `/api/v1/auth`.

Se distinguen dos ámbitos:

1. **Público**, sin aplicación de middleware, que expone:
  - `POST /api/v1/auth/register` → `RegisterHandler`
  - `POST /api/v1/auth/login` → `LoginHandler`
2. **Protegido**, con middleware JWT (`AuthMiddleware`), que añade:
  - `POST /api/v1/auth/logout` → `LogoutHandler`
  - `GET /api/v1/auth/validate` → `ValidateHandler`

El middleware se aplica en bloque al grupo protegido, garantizando la validación del token y la extracción del contexto de sesión antes de invocar los handlers.

```
1 // SetupAuthRoutes configura las rutas relacionadas con la autenticación
2 func SetupAuthRoutes(app *fiber.App, authController *controllers2.AuthController, sessionRepo repositories.SessionRepository) {
3     // Rutas públicas de autenticación
4     authGroup := app.Group("/api/v1/auth")
5     authGroup.Post("/register", authController.RegisterHandler)
6     authGroup.Post("/login", authController.LoginHandler)
7
8     // Rutas protegidas con JWT
9     protected := authGroup.Group("/", middlewares.AuthMiddleware(sessionRepo))
10    protected.Post("/logout", authController.LogoutHandler)
11    protected.Get("/validate", authController.ValidateHandler)
12 }
13 }
```



## Rutas de usuario

Definidas en [api/user\\_routes.go](#), todas protegidas mediante [AuthMiddleware](#) desde el prefijo [/api/v1/user](#):

- *GET /api/v1/user/me → MeHandler*
- *DELETE /api/v1/user/delete → DeleteHandler*



```

1 // SetupUserRoutes configura las rutas relacionadas con el usuario autenticado
2 func SetupUserRoutes(app *fiber.App, userController *controllers.UserController, sessionRepo
3                      repositories.SessionRepository) {
4     // Rutas del usuario autenticado
5     userGroup := app.Group("/api/v1/user", middlewares.AuthMiddleware(sessionRepo))
6     userGroup.Get("/me", userController.MeHandler)
7     userGroup.Delete("/delete", userController.DeleteHandler)
8 }
```

Este agrupamiento asegura que únicamente las solicitudes autenticadas pueden acceder a información personal y gestionar la propia cuenta, simplificando la responsabilidad de los controladores y alineándose con el principio de responsabilidad única.

### 5.1.10. Middleware previo a los controladores

El middleware de autenticación se implementa en el paquete [presentation/middlewares](#) bajo la función [AuthMiddleware](#), que recibe una instancia de [SessionRepository](#) para verificar la validez de la sesión asociada al token JWT. Su funcionamiento es el siguiente:

#### 1. Extracción del encabezado

Se obtiene el valor del encabezado `Authorization`. Si no está presente, se devuelve un error 401 indicando “*Token no proporcionado*”.

#### 2. Validación de formato

Se comprueba que el encabezado comience con el prefijo “*Bearer*”; de lo contrario, se retorna un error 401 con mensaje “*Formato de token inválido*”.



### 3. Parseo y verificación del JWT

Se elimina el prefijo y se invoca `security.ValidateToken(token)`, que reconstruye los claims y valida la firma y la expiración. En caso de fallo, se responde con un 401 y “*Token inválido o expirado*”.

### 4. Comprobación de sesión activa

Con el identificador único del token (`claims.ID`), se consulta el repositorio de sesiones mediante `sessionRepo.GetSessionByJwtID(claims.ID)`. Si no existe registro u ocurre un error, se retorna un 401 indicando “*Sesión inválida o terminada*”.

### 5. Contextualización de la petición

Si todas las comprobaciones son exitosas, se almacenan en el contexto de Fiber las variables `userID` y `jwtID`, extraídas de los claims, usando `c.Locals(...)`. A continuación, se invoca `c.Next()`, permitiendo que la ejecución prosiga hacia el controlador.



```

1 func AuthMiddleware(sessionRepo repositories.SessionRepository) fiber.Handler {
2     return func(c *fiber.Ctx) error {
3         auth := c.Get("Authorization")
4         if auth == "" {
5             return response.Error(c, fiber.StatusUnauthorized, "Token no proporcionado")
6         }
7         const prefix = "Bearer "
8         if !strings.HasPrefix(auth, prefix) {
9             return response.Error(c, fiber.StatusUnauthorized, "Formato de token inválido")
10        }
11        token := strings.TrimPrefix(auth, prefix)
12        claims, err := security.ValidateToken(token)
13        if err != nil {
14            return response.Error(c, fiber.StatusUnauthorized, "Token inválido o expirado")
15        }
16        sess, err := sessionRepo.GetSessionByJwtID(claims.ID)
17        if err != nil || sess == nil {
18            return response.Error(c, fiber.StatusUnauthorized, "Sesión inválida o terminada")
19        }
20        c.Locals("userID", claims.Sub)
21        c.Locals("jwtID", claims.ID)
22        return c.Next()
23    }
24 }

```

Con este middleware, todos los endpoints protegidos restringen el acceso a usuarios con un JWT válido y una sesión activa en la base de datos, garantizando la seguridad de las rutas antes de alcanzar la lógica de negocio en los controladores.



### 5.1.11. Mensaje de manejo de errores

El paquete `response` centraliza el formato de todas las respuestas HTTP y proporciona utilidades para estandarizar tanto los casos de éxito como los de error.

Se define la estructura genérica `Response` con tres campos:

- **Success (bool)**: indica si la operación fue satisfactoria.
- **Message (string)**: descripción breve del resultado o del error.
- **Data (interface{})**: payload opcional con datos adicionales en casos de éxito.

La función interna `Send` recibe el contexto de Fiber, el código de estado HTTP, el indicador de éxito, el mensaje y los datos; construye la instancia de `Response` y la serializa a JSON con el estatus correspondiente.

A partir de `Send` se ofrecen dos atajos:

- **Success(c, status, message, data)** para respuestas exitosas (códigos 2xx), estableciendo `Success` a `true` y permitiendo incluir un objeto `data`.
- **Error(c, status, message)** para respuestas de error, fijando `Success` a `false` y omitiendo el campo `Data`.

```

● ● ●

1 // Response es la forma genérica de todas las respuestas.
2 type Response struct {
3     Success bool      `json:"success"`
4     Message string    `json:"message"`
5     Data     interface{} `json:"data,omitempty"`
6 }
7
8 // Send es la función interna que envía la respuesta.
9 func Send(c *fiber.Ctx, status int, success bool, message string, data interface{}) error {
10     return c.Status(status).JSON(Response{
11         Success: success,
12         Message: message,
13         Data:     data,
14     })
15 }
16
17 // Success shortcut para respuestas 2xx
18 func Success(c *fiber.Ctx, status int, message string, data interface{}) error {
19     return Send(c, status, true, message, data)
20 }
21
22 // Error shortcut para respuestas de error
23 func Error(c *fiber.Ctx, status int, message string) error {
24     return Send(c, status, false, message, nil)
25 }

```



Este enfoque garantiza uniformidad en la forma de comunicar resultados y errores en toda la API, facilita la evolución del esquema de respuesta y mejora la experiencia de cliente al consumir los endpoints.

### 5.1.2. Funcionamiento de la aplicación

El flujo de ejecución de cada acción en la aplicación se organiza siguiendo la arquitectura en tres capas (presentación, dominio, persistencia y eventos) y haciendo uso de middleware para garantizar la autorización de las operaciones protegidas. A continuación, se ofrece una visión general de cómo interactúan estas capas en cada caso de uso:

- Desde la capa de presentación, un controlador recibe la petición HTTP y, tras la validación de seguridad (si procede), extrae los datos necesarios y delega en el servicio de dominio correspondiente.
- En la capa de dominio, el servicio aplica la lógica de negocio: validaciones de reglas, generación de datos (por ejemplo, token JWT o identificadores de sesión), interacción con repositorios para persistir o consultar entidades y emisión de eventos.
- Los repositorios encapsulan el acceso a la base de datos mediante GORM y, cuando se requiera, validan la existencia o el estado de un recurso antes de realizar operaciones de escritura o lectura.
- Para acciones que implican comunicación asíncrona, los servicios publican eventos en RabbitMQ a través del *EventPublisher*, desacoplando así la lógica principal de tareas secundarias (notificaciones, auditoría).
- Finalmente, el controlador convierte el resultado del servicio en una respuesta estandarizada usando el paquete de manejo de respuestas, cerrando el ciclo con un JSON uniforme que indica éxito o fracaso y, cuando aplica, devuelve los datos resultantes.

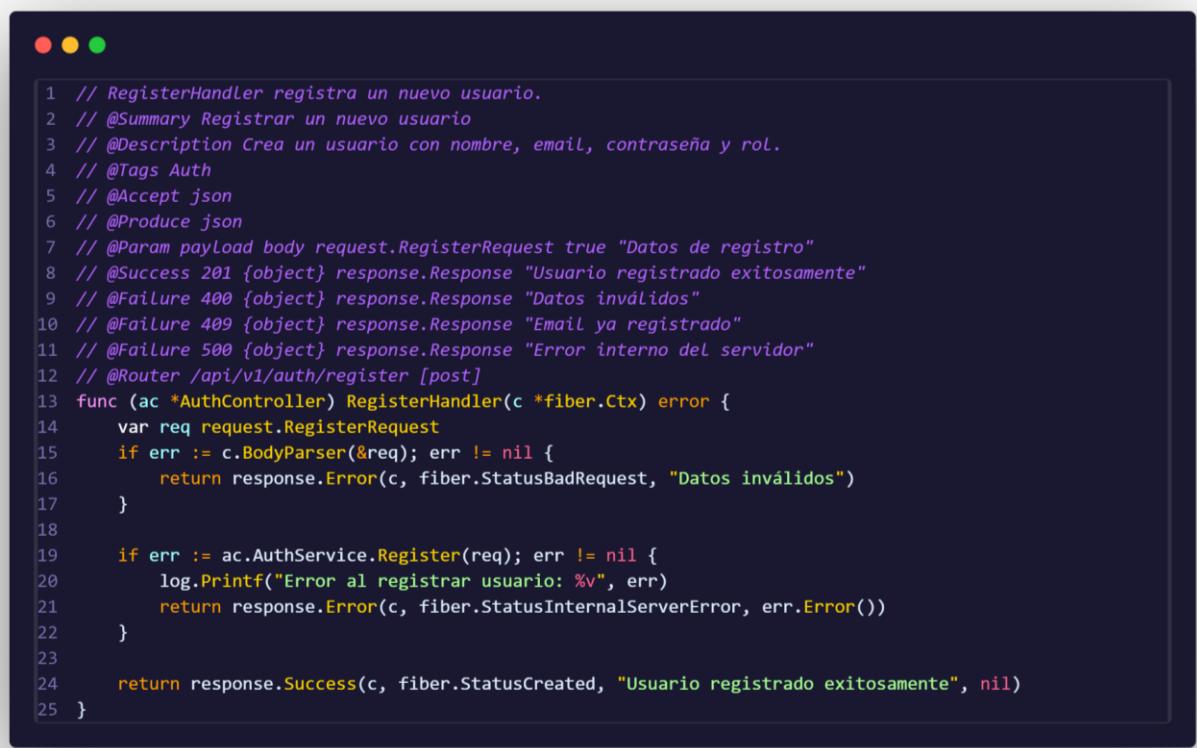
A continuación, se presentan, apoyados en diagramas, los flujos de invocación entre métodos para cada caso de uso. Además, en el código fuente se incluyen comentarios explicativos que facilitan la comprensión de cada parte del proceso y ayudan a seguir la lógica detrás de cada acción.



### 5.1.12.1. Registrar un usuario

Es el proceso mediante el cual un nuevo usuario es creado en el sistema. Consiste en recibir los datos personales (nombre, correo, contraseña), validarlos y almacenarlos en la base de datos. La contraseña se guarda de forma segura mediante hash, y al finalizar, se puede emitir un token JWT para autenticar al nuevo usuario.

#### Controlador → Servicio



```
1 // RegisterHandler registra un nuevo usuario.
2 // @Summary Registrar un nuevo usuario
3 // @Description Crea un usuario con nombre, email, contraseña y rol.
4 // @Tags Auth
5 // @Accept json
6 // @Produce json
7 // @Param payload body request.RegisterRequest true "Datos de registro"
8 // @Success 201 {object} response.Response "Usuario registrado exitosamente"
9 // @Failure 400 {object} response.Response "Datos inválidos"
10 // @Failure 409 {object} response.Response "Email ya registrado"
11 // @Failure 500 {object} response.Response "Error interno del servidor"
12 // @Router /api/v1/auth/register [post]
13 func (ac *AuthController) RegisterHandler(c *fiber.Ctx) error {
14     var req request.RegisterRequest
15     if err := c.BodyParser(&req); err != nil {
16         return response.Error(c, fiber.StatusBadRequest, "Datos inválidos")
17     }
18
19     if err := ac.AuthService.Register(req); err != nil {
20         log.Printf("Error al registrar usuario: %v", err)
21         return response.Error(c, fiber.StatusInternalServerError, err.Error())
22     }
23
24     return response.Success(c, fiber.StatusCreated, "Usuario registrado exitosamente", nil)
25 }
```

#### Servicio → Repositorio



```

1 // Register crea un nuevo usuario en la base de datos.
2 func (as *AuthServiceImpl) Register(req request.RegisterRequest) error {
3     // Validación de campos requeridos
4     if req.Email == "" || req.Password == "" || req.Name == "" {
5         return fmt.Errorf("todos los campos son requeridos")
6     }
7
8     // Validación de formato de correo electrónico
9     if !email.IsValidEmail(req.Email) {
10        return fmt.Errorf("el correo electrónico no es válido")
11    }
12
13    // Validación de rol
14    validRoles := map[models2.UserRole]struct{}{
15        models2.Admin:      {},
16        models2.Instructor: {},
17        models2.Student:    {},
18    }
19
20    // Verificamos si el rol es válido
21    if _, valid := validRoles[models2.UserRole(req.Role)]; !valid {
22        return fmt.Errorf("rol inválido: %s", req.Role)
23    }
24
25    // Hash de la contraseña
26    hashedPassword, err := security.HashPassword(req.Password)
27    if err != nil {
28        return fmt.Errorf("error al procesar la contraseña: %v", err)
29    }
30
31    // Crear el objeto user
32    user := models2.User{
33        Name:      req.Name,
34        Email:     req.Email,
35        PasswordHash: hashedPassword,
36        Role:      models2.UserRole(req.Role),
37    }
38
39    // Registrar al usuario en el repositorio
40    if err := as.UserRepo.Create(&user); err != nil {
41        return fmt.Errorf("error al registrar el usuario: %v", err)
42    }
43
44    return nil
45 }

```

### Repositorio → Base de datos

```

1 // Create guarda un nuevo usuario en la base de datos.
2 func (r *GormUserRepository) Create(user *models.User) error {
3     return r.db.Create(user).Error
4 }

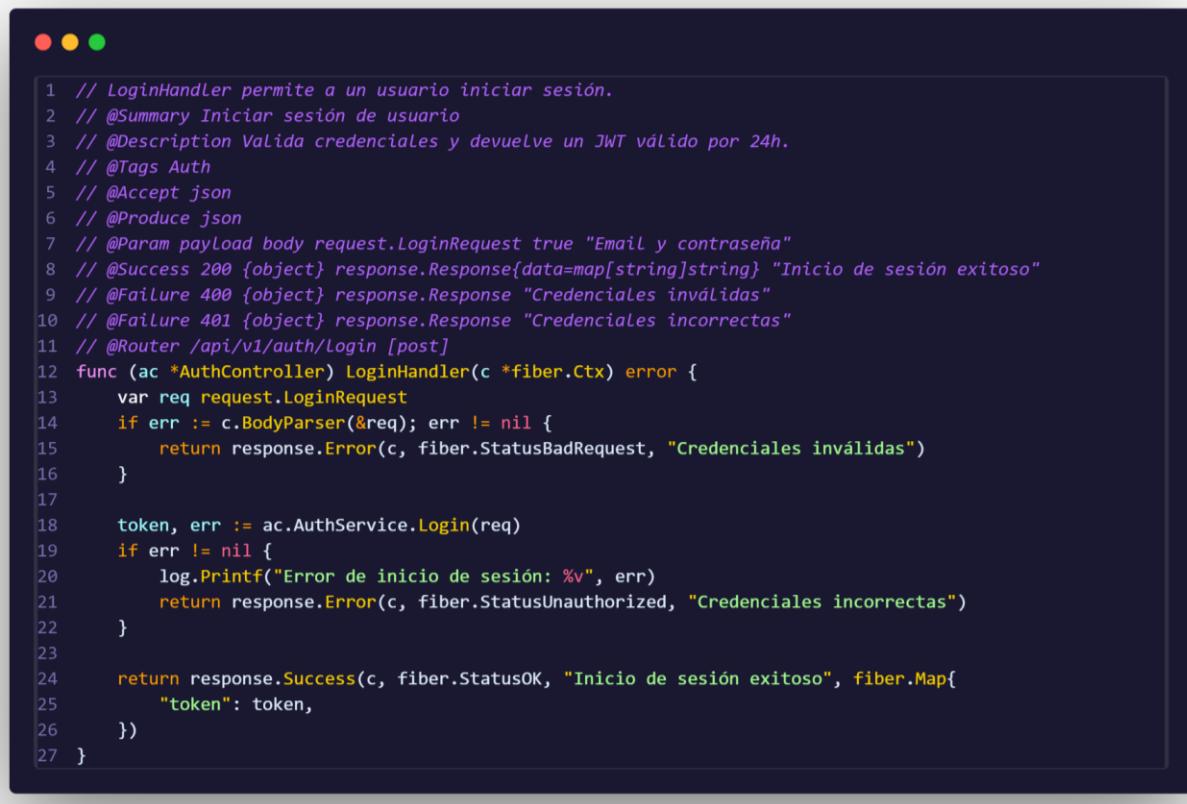
```



### 5.1.12.2. Iniciar sesión

Es el proceso de autenticación en el que un usuario existente proporciona sus credenciales (correo y contraseña). El sistema valida los datos, genera un token JWT y crea una nueva sesión asociada al usuario. Este token será utilizado para autenticar futuras solicitudes.

Controlador → Servicio



```
1 // LoginHandler permite a un usuario iniciar sesión.
2 // @Summary Iniciar sesión de usuario
3 // @Description Valida credenciales y devuelve un JWT válido por 24h.
4 // @Tags Auth
5 // @Accept json
6 // @Produce json
7 // @Param payload body request.LoginRequest true "Email y contraseña"
8 // @Success 200 {object} response.Response{data=map[string]string} "Inicio de sesión exitoso"
9 // @Failure 400 {object} response.Response "Credenciales inválidas"
10 // @Failure 401 {object} response.Response "Credenciales incorrectas"
11 // @Router /api/v1/auth/login [post]
12 func (ac *AuthController) LoginHandler(c *fiber.Ctx) error {
13     var req request.LoginRequest
14     if err := c.BodyParser(&req); err != nil {
15         return response.Error(c, fiber.StatusBadRequest, "Credenciales inválidas")
16     }
17
18     token, err := ac.AuthService.Login(req)
19     if err != nil {
20         log.Printf("Error de inicio de sesión: %v", err)
21         return response.Error(c, fiber.StatusUnauthorized, "Credenciales incorrectas")
22     }
23
24     return response.Success(c, fiber.StatusOK, "Inicio de sesión exitoso", fiber.Map{
25         "token": token,
26     })
27 }
```



Servicio → Repositorio

```

● ● ●

1 // Login auténtica un usuario y crea una sesión con un token JWT.
2 func (as *AuthService) Login(req request.LoginRequest) (string, error) {
3     // Verifica que el usuario exista por correo electrónico
4     user, err := as.UserRepo.FindByEmail(req.Email)
5     if err != nil {
6         return "", fmt.Errorf("usuario no encontrado: %v", err)
7     }
8     // Verifica que la contraseña sea correcta
9     if !security.CheckPasswordHash(req.Password, user.PasswordHash) {
10         return "", errors.New("credenciales inválidas")
11     }
12
13     jwtID := uuid.New()
14     // Genera un nuevo JWT para el usuario
15     token, err := security.GenerateToken(user.ID, string(user.Role), jwtID.String(), 24*time.Hour)
16     if err != nil {
17         return "", fmt.Errorf("error al generar el token: %v", err)
18     }
19
20     session := models2.Session{
21         UserID:    user.ID,
22         JwtID:    jwtID,
23         IssuedAt: time.Now(),
24         ExpiresAt: time.Now().Add(24 * time.Hour),
25     }
26
27     // Guardamos la sesión en el repositorio
28     if err := as.SessionRepo.CreateSession(&session); err != nil {
29         return "", fmt.Errorf("error al crear sesión: %v", err)
30     }
31
32     return token, nil
33 }

```

Servicio → Utilidad para comprobar contraseñas

```

● ● ●

1 // CheckPasswordHash compara una contraseña sin procesar con su hash y devuelve si coinciden.
2 func CheckPasswordHash(password, hash string) bool {
3     // Compara la contraseña sin procesar con el hash
4     err := bcrypt.CompareHashAndPassword([]byte(hash), []byte(password))
5
6     return err == nil
7 }
8

```



Repositorio de usuario → Base de datos

```

● ● ●
1 // FindByEmail busca un usuario por su correo electrónico.
2 func (r *GormUserRepository) FindByEmail(email string) (*models.User, error) {
3     var user models.User
4     err := r.db.Where("email = ?", email).First(&user).Error
5     return &user, err
6 }

```

Repositorio de sesión → Base de datos

```

● ● ●
1 // CreateSession guarda una nueva sesión en la base de datos.
2 func (r *GormSessionRepository) CreateSession(session *models.Session) error {
3     return r.DB.Create(session).Error
4 }

```

**5.1.12.3. Cerrar sesión**

Es la acción que finaliza una sesión activa del usuario. Se invalida el token JWT actual eliminando o marcando como terminada la sesión correspondiente en la base de datos, impidiendo su reutilización.

Controlador → Servicio

```

● ● ●
1 // LogoutHandler cierra la sesión del usuario actual.
2 // @Summary Cerrar sesión
3 // @Description Invalida el JWT actual eliminando la sesión en BD.
4 // @Tags Auth
5 // @Security BearerAuth
6 // @Accept json
7 // @Produce json
8 // @Success 200 {object} response.Response "Sesión cerrada exitosamente"
9 // @Failure 401 {object} response.Response "Token no proporcionado o inválido"
10 // @Failure 500 {object} response.Response "Error interno al cerrar sesión"
11 // @Router /api/v1/auth/logout [post]
12 func (ac *AuthController) LogoutHandler(c *fiber.Ctx) error {
13     jwtID := c.Locals("jwtID").(string)
14     if err := ac.AuthService.Logout(jwtID); err != nil {
15         log.Printf("Error al cerrar sesión: %v", err)
16         return response.Error(c, fiber.StatusInternalServerError, "Error al cerrar sesión")
17     }
18     return response.Success(c, fiber.StatusOK, "Sesión cerrada exitosamente", nil)
19 }

```

Servicio → Repositorio

```

1 // Logout elimina la sesión asociada a un JWT específico.
2 func (as *AuthServiceImpl) Logout(jwtID string) error {
3     // Llamamos al repositorio para eliminar la sesión usando el jwtID
4     if err := as.SessionRepo.DeleteSessionByJwtID(jwtID); err != nil {
5         return fmt.Errorf("error al eliminar la sesión: %v", err)
6     }
7     return nil
8 }
```

Repositorio → Base de datos

```

1 // DeleteSessionByJwtID elimina una sesión específica por su JwtID.
2 func (r *GormSessionRepository) DeleteSessionByJwtID(jwtID string) error {
3     // Elimina la sesión cuya jwtID coincida
4     return r.DB.Where("jwt_id = ?", jwtID).Delete(&models.Session{}).Error
5 }
```

**5.1.12.4. Validar la sesión de un usuario**

Permite verificar si un usuario ha iniciado sesión correctamente mediante un token de autenticación (por ejemplo, JWT). Esta acción comprueba que el token proporcionado sea válido, no haya expirado, y esté asociado a un usuario existente.

Controlador → Servicio

```

1 // ValidateHandler valida el token JWT y devuelve email y tiempo restante.
2 // @Summary Validar token JWT
3 // @Description Comprueba que el token está activo y devuelve email y tiempo restante.
4 // @Tags Auth
5 // @Security BearerAuth
6 // @Accept json
7 // @Produce json
8 // @Success 200 {object} response.Response{data=map[string]string} "Token válido"
9 // @Failure 401 {object} response.Response "Token inválido o expirado"
10 // @Router /api/v1/auth/validate [get]
11 func (ac *AuthController) ValidateHandler(c *fiber.Ctx) error {
12     userID := c.Locals("userID").(string)
13     user, remaining, err := ac.AuthService.Validate(userID)
14     if err != nil {
15         log.Printf("Error de validación de token: %v", err)
16         return response.Error(c, fiber.StatusUnauthorized, "Token inválido o sesión expirada")
17     }
18
19     expiresIn := fmt.Sprintf("%02d:%02dm", int(remaining.Hours()), int(remaining.Minutes()%60))
20     return response.Success(c, fiber.StatusOK, "Token válido", fiber.Map{
21         "email": user.Email,
22         "expires_in": expiresIn,
23     })
24 }
25 }
```



Servicio → Repositorio

```

1 // Valida devuelve información del usuario si el ID es válido.
2 func (as *AuthService) Validate(userID string) (*models.User, time.Duration, error) {
3     // Buscamos el usuario
4     user, err := as.UserRepo.FindByID(userID)
5     if err != nil {
6         return nil, 0, fmt.Errorf("usuario no encontrado: %v", err)
7     }
8
9     // Recuperamos la sesión activa (solo las que no hayan expirado)
10    session, err := as.SessionRepo.FindActiveSessionByUserID(userID)
11    if err != nil {
12        return nil, 0, fmt.Errorf("sesión no encontrada o expirada: %v", err)
13    }
14
15    // Calculamos el tiempo restante
16    remaining := session.ExpiresAt.Sub(time.Now())
17    if remaining < 0 {
18        return nil, 0, fmt.Errorf("la sesión ya expiró")
19    }
20
21    return user, remaining, nil
22 }
23

```

Repositorio de usuarios → Base de datos

```

1 // FindByID busca un usuario por su ID.
2 func (r *GormUserRepository) FindByID(userID string) (*models.User, error) {
3     var user models.User
4     err := r.db.First(&user, "id = ?", userID).Error
5     return &user, err
6 }
7

```

Repositorio de sesión → Base de datos

```

1 // FindActiveSessionByUserID encuentra una sesión activa por el ID del usuario.
2 func (r *GormSessionRepository) FindActiveSessionByUserID(userID string) (*models.Session, error) {
3     var session models.Session
4     err := r.DB.Where("user_id = ? AND expires_at > ?", userID, time.Now()).First(&session).Error
5     if err != nil {
6         return nil, err
7     }
8     return &session, nil
9 }

```



### 5.1.12.5. Ver perfil de un usuario

Permite a un usuario autenticado acceder a su información personal registrada en el sistema.

#### Controlador → Servicio

```

● ● ●

1 // MeHandler obtiene información del usuario actual.
2 // @Summary Obtener información del usuario actual
3 // @Description Devuelve los datos del usuario autenticado.
4 // @Tags User
5 // @Security BearerAuth
6 // @Accept json
7 // @Produce json
8 // @Success 200 {object} response.Response{data=map[string]interface{}} "Información del usuario"
9 // @Failure 401 {object} response.Response "No autorizado"
10 // @Failure 500 {object} response.Response "Error interno del servidor"
11 // @Router /api/v1/auth/me [get]
12 func (uc *UserController) MeHandler(c *fiber.Ctx) error {
13     userID := c.Locals("userID").(string)
14     user, err := uc.UserService.GetMe(userID)
15     if err != nil {
16         log.Printf("Error al obtener la información del usuario: %v", err)
17         return response.Error(c, fiber.StatusInternalServerError, "No se pudo obtener el usuario")
18     }
19     return response.Success(c, fiber.StatusOK, "Información del usuario", fiber.Map{
20         "user": user,
21     })
22 }
```

#### Servicio → Repositorio

```

● ● ●

1 // GetMe obtiene la información del usuario actual por su ID.
2 func (us *UserService) GetMe(userID string) (*models.User, error) {
3     // Buscar el usuario por ID
4     user, err := us.UserRepo.FindByID(userID)
5     if err != nil {
6         // Envolver el error con más contexto para facilitar el rastreo
7         return nil, fmt.Errorf("no se pudo obtener el usuario con ID %s: %w", userID, err)
8     }
9     return user, nil
10 }
```

#### Repositorio → Base de datos

```

● ● ●

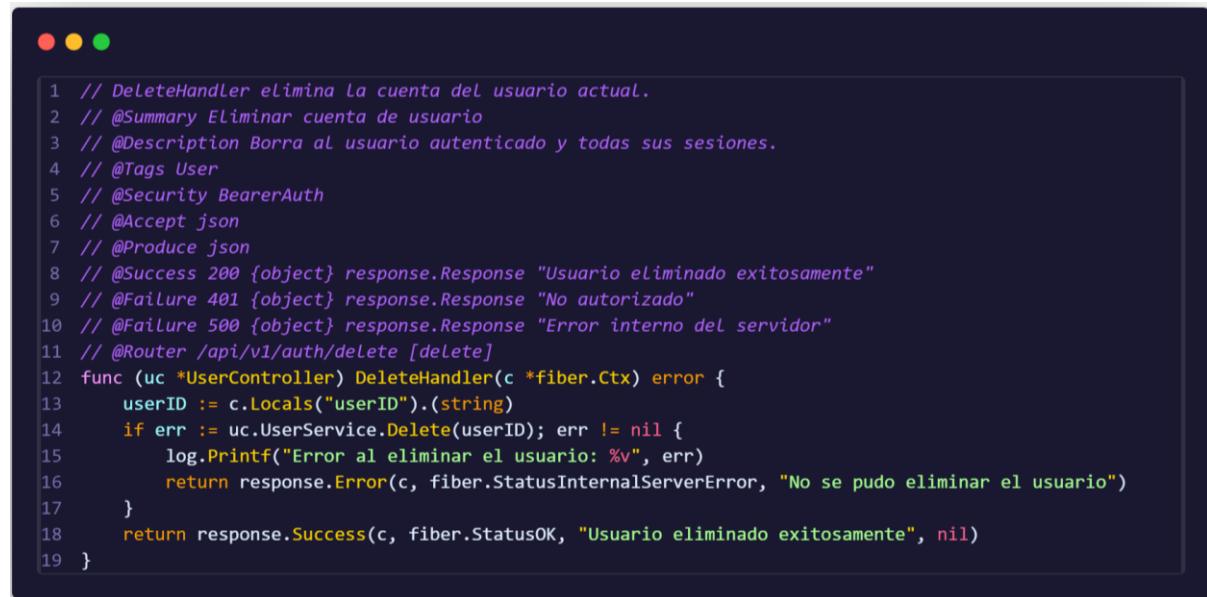
1 // FindByID busca un usuario por su ID.
2 func (r *GormUserRepository) FindByID(userID string) (*models.User, error) {
3     var user models.User
4     err := r.db.First(&user, "id = ?", userID).Error
5     return &user, err
6 }
```



### 5.1.12.6. Eliminar un usuario

Es una acción irreversible que elimina la cuenta del usuario autenticado, incluyendo su información personal y todas sus sesiones activas. El sistema emite un evento para notificar a otros servicios y asegura que la eliminación esté protegida mediante autenticación.

#### Controlador → Servicio

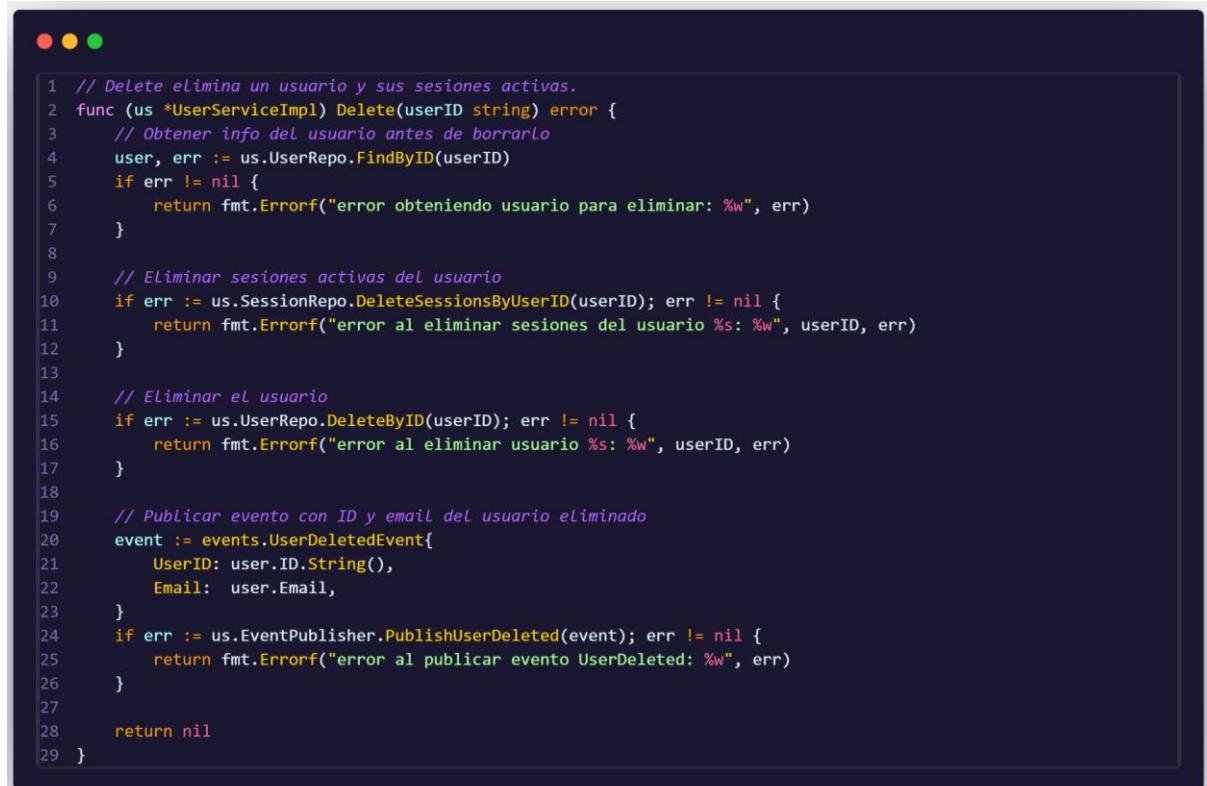


```

1 // DeleteHandler elimina la cuenta del usuario actual.
2 // @Summary Eliminar cuenta de usuario
3 // @Description Borra al usuario autenticado y todas sus sesiones.
4 // @Tags User
5 // @Security BearerAuth
6 // @Accept json
7 // @Produce json
8 // @Success 200 {object} response.Response "Usuario eliminado exitosamente"
9 // @Failure 401 {object} response.Response "No autorizado"
10 // @Failure 500 {object} response.Response "Error interno del servidor"
11 // @Router /api/v1/auth/delete [delete]
12 func (uc *UserController) DeleteHandler(c *fiber.Ctx) error {
13     userID := c.Locals("userID").(string)
14     if err := uc.UserService.Delete(userID); err != nil {
15         log.Printf("Error al eliminar el usuario: %v", err)
16         return response.Error(c, fiber.StatusInternalServerError, "No se pudo eliminar el usuario")
17     }
18     return response.Success(c, fiber.StatusOK, "Usuario eliminado exitosamente", nil)
19 }

```

#### Servicio → Evento + Repositorio



```

1 // Delete elimina un usuario y sus sesiones activas.
2 func (us *UserServiceImpl) Delete(userID string) error {
3     // Obtener info del usuario antes de borrarlo
4     user, err := us.UserRepo.FindByID(userID)
5     if err != nil {
6         return fmt.Errorf("error obteniendo usuario para eliminar: %w", err)
7     }
8
9     // Eliminar sesiones activas del usuario
10    if err := us.SessionRepo.DeleteSessionsByUserID(userID); err != nil {
11        return fmt.Errorf("error al eliminar sesiones del usuario %s: %w", userID, err)
12    }
13
14    // Eliminar el usuario
15    if err := us.UserRepo.DeleteByID(userID); err != nil {
16        return fmt.Errorf("error al eliminar usuario %s: %w", userID, err)
17    }
18
19    // Publicar evento con ID y email del usuario eliminado
20    event := events.UserDeletedEvent{
21        UserID: user.ID.String(),
22        Email: user.Email,
23    }
24    if err := us.EventPublisher.PublishUserDeleted(event); err != nil {
25        return fmt.Errorf("error al publicar evento UserDeleted: %w", err)
26    }
27
28    return nil
29 }

```



## Repositorio de usuario → Base de datos

```

1 // DeleteByID elimina un usuario por su ID.
2 func (r *GormUserRepository) DeleteByID(userID string) error {
3     return r.db.Where("id = ?", userID).Delete(&models.User{}).Error
4 }

1 // FindByID busca un usuario por su ID.
2 func (r *GormUserRepository) FindByID(userID string) (*models.User, error) {
3     var user models.User
4     err := r.db.First(&user, "id = ?", userID).Error
5     return &user, err
6 }

```

## Evento → RabbitMQ

```

1 // PublishUserDeleted publica un evento cuando un usuario es eliminado
2 func (p *EventPublisher) PublishUserDeleted(event events.UserDeletedEvent) error {
3     // Crear el evento de usuario eliminado
4     body, err := json.Marshal(struct {
5         Event string `json:"event"`
6         Data   events.UserDeletedEvent `json:"data"`
7     }{
8         Event: "UserDeleted",
9         Data:  event,
10    })
11    if err != nil {
12        return fmt.Errorf("error serializando evento: %w", err)
13    }
14
15    // Publicar el evento en RabbitMQ
16    err = p.channel.Publish(
17        "golearnix.events", // exchange
18        "user.deleted",    // routing key
19        false,             // mandatory
20        false,             // immediate
21        amqp091.Publishing{
22            ContentType: "application/json",
23            Body:        body,
24        },
25    )
26
27    if err != nil {
28        return fmt.Errorf("error publicando evento: %w", err)
29    }
30
31    return nil
32 }

```

El modelo del evento `UserDeletedEvent` incluye el id del usuario y su email.

```

1 type UserDeletedEvent struct {
2     UserID string `json:"user_id"`
3     Email  string `json:"email"`
4 }

```



Componentes del evento publicado:

- **Exchange: "golearnix.events"**

El exchange es un componente de RabbitMQ que recibe los mensajes publicados y los distribuye a las colas según la routing key. En este caso, el exchange usado se llama golearnix.events.

- **Routing key: "user.deleted"**

Esta clave identifica el tipo de evento que se está publicando. Aquí se indica que un usuario ha sido eliminado. Los consumidores que estén suscritos a esta clave recibirán el mensaje.

- **amqp091.Publishing{...}**

Este bloque define el contenido del mensaje:

- **ContentType: "application/json"**: indica que el cuerpo del mensaje está en formato JSON.
- **Body: body**: contiene los datos del evento (**UserDeletedEvent**).
- **false, false** (parámetros mandatory e immediate)
  - **mandatory**: si es false, no se requiere que el mensaje llegue a alguna cola.
  - **immediate**: si es false, el mensaje se puede encolar incluso si no hay consumidores activos en ese momento.

Estas propiedades están definidas en el método PublishUserDeleted:

```

1 // Publicar el evento en RabbitMQ
2     err = p.channel.Publish(
3         "golearnix.events", // exchange
4         "user.deleted",    // routing key
5         false,             // mandatory
6         false,             // immediate
7         amqp091.Publishing{
8             ContentType: "application/json",
9             Body:        body,
10            },
11        )

```



## 5.2. Aplicación con Java para la gestión de cursos

Este microservicio ha sido desarrollado en Java empleando el framework Spring Boot. Su función principal consiste en la gestión integral del catálogo de cursos, incluyendo la creación, lectura, actualización y eliminación (CRUD) de los mismos, así como la asignación de instructores y el seguimiento del progreso de los alumnos.

### 5.2.1. ¿Por qué usar Java con Spring Boot?

Java, junto con Spring Boot, ha sido la combinación escogida por los motivos que ya se exponen con más detalle en el marco conceptual:

- **Portabilidad y madurez de la JVM**

Java se ejecuta sobre la Máquina Virtual de Java (JVM), lo que garantiza que la misma aplicación pueda desplegarse sin cambios en cualquier plataforma con soporte JVM. Su largo recorrido industrial avala la estabilidad en entornos de producción y la disponibilidad de soluciones de optimización de rendimiento.

- **Autoconfiguración y arranque rápido**

Spring Boot proporciona un modelo de “*convention over configuration*” que simplifica enormemente el arranque y la configuración de nuevos proyectos. Mediante dependencias “*starter*” y propiedades predefinidas, es posible levantar un servicio RESTf con escasos ficheros de configuración y tiempos de inicio reducidos.

- **Ecosistema rico y modularidad**

El ecosistema de Spring incluye proyectos especializados (Spring Data, Spring Security, Spring Cloud, etc.) que facilitan la integración de funcionalidades avanzadas, como persistencia en bases de datos relacionales o NoSQL, autenticación/autorización, tolerancia a fallos y descubrimiento de servicios, manteniendo una arquitectura desacoplada y fácilmente testeable.



- **Gestión de dependencias y empaquetado**

Gracias a Maven o Gradle, los ciclos de compilación, gestión de versiones y despliegue se orquestan de forma uniforme. Spring Boot incluye empaquetado “*fat-jar*”, lo que permite distribuir cada microservicio como un único ejecutable que incorpora el contenedor embebido (Tomcat) simplificando la entrega y la operación.

- **Seguridad y monitorización integradas**

Con Spring Security es posible implantar mecanismos de autenticación y autorización basados en JWT, OAuth 2.0 o LDAP con una configuración mínima. Spring Boot Actuator, por su parte, ofrece endpoints listos para exponer métricas, salud del sistema y trazas, facilitando la observabilidad y la integración con plataformas de monitorización como Prometheus o ELK.

- **Comunidad activa y soporte empresarial**

Spring Boot cuenta con una gran comunidad de desarrolladores y el respaldo, lo que asegura actualizaciones frecuentes, correcciones de seguridad y abundante documentación, tutoriales y ejemplos de casos de uso reales en grandes organizaciones.

En conjunto, Java con Spring Boot proporciona un entorno sólido, estándar de la industria, y con un conjunto de herramientas que aceleran el desarrollo, garantizan la calidad del código y facilitan el mantenimiento evolutivo de los microservicios dedicados a la gestión de cursos.

### 5.2.2. Arquitectura Hexagonal

La arquitectura hexagonal (o Ports & Adapters) es una variante de la Clean Architecture que sitúa el dominio en el centro del sistema y lo aísla completamente de las capas exteriores. Para lograrlo, define dos tipos de elementos:

- **Puertos (Ports):** Interfaces que describen los contratos públicos de entrada y salida del dominio.
- **Adaptadores (Adapters):** Implementaciones concretas de estos puertos para integrar el dominio con tecnologías externas (bases de datos, colas de mensajes, APIs REST, etc.).

De este modo, las dependencias siempre fluyen hacia el interior (hacia el núcleo del dominio) y la lógica de negocio permanece independiente de cualquier elección tecnológica.

### 5.2.2.1. ¿Por qué usar la arquitectura hexagonal en este proyecto?

Se ha escogido principalmente por 4 ventajas:

- **Independencia tecnológica:** El dominio no conoce detalles de la infraestructura, lo que facilita cambiar de base de datos, sistema de mensajería o framework sin tocar la lógica de negocio.
- **Inversión de dependencias:** Al depender de interfaces, no de implementaciones, es posible sustituir adaptadores (por ejemplo, un repositorio PostgreSQL por otro Redis) sin afectar al código del dominio.
- **Separación de responsabilidades:** Cada capa (dominio, aplicación e infraestructura) tiene una única responsabilidad, mejorando la mantenibilidad y la claridad del diseño.
- **Facilita el DDD:** El enfoque encaja de forma natural con Domain-Driven Design, pues promueve un núcleo de dominio puro, rodeado de adaptadores que traducen los eventos externos a conceptos del dominio.

### 5.2.2.2. ¿Por qué crear un proyecto multimódulo con Maven?

La decisión de estructurar el proyecto como un conjunto de módulos gestionados por Maven responde al objetivo de maximizar la cohesión interna y minimizar el acoplamiento entre componentes. Cada módulo asume una responsabilidad bien definida, desde la exposición de la API REST hasta la persistencia de datos o la inicialización de caché, lo que refuerza la separación de preocupaciones y facilita la comprensión del alcance de cada parte del sistema.

Al mismo tiempo, esta modularidad potencia la reutilización: otros microservicios pueden incorporar únicamente aquellos módulos que les resulten pertinentes, como el dominio, sin verse obligados a arrastrar dependencias superfluas.

Maven, además, optimiza los ciclos de construcción mediante compilación incremental, de modo que solo recompila los módulos modificados, lo que acelera sobremanera los despliegues y las pruebas.



Por último, la gestión centralizada de dependencias en un único pom.xml padre garantiza la consistencia de versiones y ajustes compartidos, reduce la duplicidad de configuración y evita posibles conflictos, favoreciendo un mantenimiento más ágil y seguro.

### 5.2.2.3. Funcionamiento

El flujo de ejecución comienza cuando un cliente emite una petición, por ejemplo, una llamada HTTP o el envío de un mensaje a RabbitMQ, que es recibida por el adaptador de entrada correspondiente.

Este adaptador se encarga de deserializar el contenido, transformándolo en un objeto de tipo comando o consulta, y lo entrega al módulo de aplicación a través del puerto de entrada definido para tal fin.

A continuación, la capa de aplicación orquesta el caso de uso concreto, invocando la lógica de negocio del dominio y, de ser necesario, delegando en los puertos de salida para persistir información o publicar eventos.

La capa de dominio, estrictamente independiente de cualquier tecnología, procesa las reglas de negocio y retorna el resultado a la capa de aplicación.

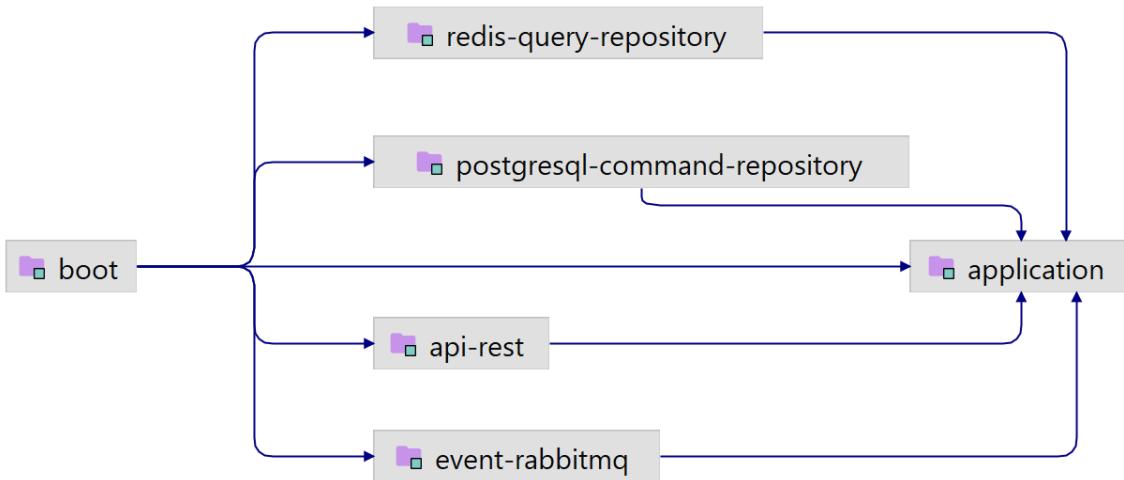
Finalmente, los adaptadores de salida implementan esos puertos para interactuar con bases de datos (PostgreSQL, Redis), colas de mensajes u otros servicios externos, mientras que el adaptador de entrada original construye la respuesta, formato JSON, confirmación de procesamiento, etc, y la envía de vuelta al cliente.

Este recorrido garantiza que todas las dependencias fluyan hacia el interior y que la lógica central permanezca aislada de los detalles de infraestructura.



#### 5.2.2.4. Estructura de carpetas/módulos

A continuación, la organización multimódulo ilustrada en la imagen del proyecto:



Cada módulo se compila y prueba de forma aislada, pero en tiempo de ejecución el módulo **boot** ensambla todos los artefactos en un único ejecutable, registrando los adaptadores y puertos bajo el contexto de Spring Boot para su correcta inyección y descubrimiento

#### 5.2.3. ¿Cómo generar el contrato Swagger a partir de los controladores?

Para exponer de forma automática un contrato OpenAPI/Swagger, se incorpora al proyecto la dependencia oficial de Springdoc OpenAPI.

Dicha librería escanea las anotaciones presentes en los controladores, como `@Tag`, `@Operation`, `@ApiResponse` y `@ApiResponses`, y pone a disposición la especificación YAML en la ruta `/v3/api-docs`, así como una interfaz interactiva de Swagger UI en `/swagger-ui.html`.

```

<!-- Dependencia de OpenAPI - Swagger -->
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
</dependency>
  
```

De este modo, la documentación generada refleja con precisión las rutas, los parámetros de entrada (`@PathVariable`, `@RequestBody`) y los posibles códigos de respuesta, todo ello sin necesidad de escribir manualmente archivos de especificación, solo a través de los controladores.

```

1  @RestController
2  @RequiredArgsConstructor
3  @RequestMapping(CourseControllerAdapter.URL)
4  @CrossOrigin(origins = "${api.cors.allowed-origins}")
5  @Tag(name = "Courses", description = "Controller used to course management")
6  public class CourseControllerAdapter {
7
8      protected static final String URL = "${api.base-path}" + "/courses";
9
10     private final CourseServicePort courseServicePort;
11     private final CurrentUserHelper currentUserHelper;
12
13     @Operation(summary = "Get all courses", description = "Retrieve a list of all available courses")
14     @ApiResponse(responseCode = "200", description = "Successful retrieval of courses")
15     @GetMapping
16     public ResponseEntity<List<Course GetAllProjection>> getAll() {
17         return ResponseEntity.ok(courseServicePort.getAll());
18     }
19
20     @Operation(summary = "Get a course by ID", description = "Retrieve a course by its ID")
21     @ApiResponsees({
22         @ApiResponse(responseCode = "200", description = "Course found"),
23         @ApiResponse(responseCode = "404", description = "Course not found", content = @Content)
24     })
25     @GetMapping("/{id}")
26     public ResponseEntity<Course> getById(@PathVariable Integer id) {
27         return ResponseEntity.ok(courseServicePort.getById(id));
28     }
29
30     @Operation(summary = "Create a new course", description = "Create a course (INSTRUCTOR role only)")
31     @ApiResponse(responseCode = "200", description = "Course successfully created")
32     @PostMapping
33     @PreAuthorize("hasRole(@role.INSTRUCTOR)")
34     public ResponseEntity<Void> create(@RequestBody @Valid Course course) {
35         courseServicePort.create(course);
36         return ResponseEntity.ok().build();
37     }
38
39     @Operation(summary = "Update a course", description = "Update an existing course by ID (INSTRUCTOR role only)")
40     @ApiResponsees({
41         @ApiResponse(responseCode = "200", description = "Course successfully updated"),
42         @ApiResponse(responseCode = "404", description = "Course not found", content = @Content)
43     })
44     @PutMapping("/{id}")
45     @PreAuthorize("hasRole(@role.INSTRUCTOR)")
46     public ResponseEntity<Void> update(@PathVariable Integer id, @RequestBody @Valid Course course) {
47         courseServicePort.update(id, course);
48         return ResponseEntity.ok().build();
49     }
50
51     @Operation(summary = "Delete a course", description = "Delete a course by ID (INSTRUCTOR role only)")
52     @ApiResponsees({
53         @ApiResponse(responseCode = "200", description = "Course successfully deleted"),
54         @ApiResponse(responseCode = "404", description = "Course not found", content = @Content)
55     })
56     @DeleteMapping("/{id}")
57     @PreAuthorize("hasRole(@role.INSTRUCTOR)")
58     public ResponseEntity<Void> delete(@PathVariable Integer id) {
59         courseServicePort.delete(id);
60         return ResponseEntity.ok().build();
61     }
62
63     @Operation(summary = "Enroll in a course", description = "Enroll current user into a course (STUDENT role only)")
64     @ApiResponse(responseCode = "200", description = "Enrollment successful")
65     @PostMapping("/{courseId}/enrollments")
66     @PreAuthorize("hasRole(@role.STUDENT)")
67     public ResponseEntity<Void> enroll(@PathVariable Integer courseId) {
68         UUID userId = currentUserHelper.getId();
69         courseServicePort.enroll(courseId, userId);
70         return ResponseEntity.ok().build();
71     }
72
73     @Operation(summary = "Complete a lesson", description = "Mark a lesson as completed by the user (STUDENT role only)")
74     @ApiResponse(responseCode = "200", description = "Lesson marked as completed")
75     @PostMapping("/{courseId}/sections/{sectionId}/lessons/{lessonId}/complete")
76     @PreAuthorize("hasRole(@role.STUDENT)")
77     public ResponseEntity<Void> completeLesson(@PathVariable Integer courseId, @PathVariable Integer sectionId, @PathVariable Integer lessonId) {
78         UUID userId = currentUserHelper.getId();
79         courseServicePort.completeLesson(courseId, sectionId, lessonId, userId);
80         return ResponseEntity.ok().build();
81     }
82
83 }

```



### 5.2.4. ¿Cómo extraer la configuración de propiedades a las variables de entorno?

La externalización de la configuración en variables de entorno se logra incorporando un fichero `.env` que contiene las claves y parámetros sensibles o variables, y haciendo que Spring Boot lo importe automáticamente al iniciar la aplicación.

En este caso, el archivo `.env` define valores como:

```
1 JWT_SECRET=9eb6dd3497ceae621c3f2e852159e992d854adcaaaac5282019d2d3353893412
2 PASS=golearnix
3 HOST=localhost
4 PSQL_PORT=5433
5 PSQL_DBNAME=course-psql-db
6 REDIS_PORT=6379
7 KFK_PORT=9092
8 RMQ_PORT=5672
9
```

Para que Spring Boot reconozca estas variables, el fichero `application.properties` incluye al comienzo la directiva:

```
1 # Fichero de variables de entorno
2 spring.config.import=file:.env[.properties]
```

De modo que todas las claves definidas en queden disponibles como propiedades del entorno. A partir de ahí, cualquier configuración convencional en puede referirse a ellas mediante la sintaxis  `${VAR_NAME};` por ejemplo, las credenciales de Flyway se especifican como `spring.flyway.user=${PASS}` y `spring.flyway.password=${PASS}`:

```
1 # Configuracion de Flyway
2 spring.flyway.enabled=true
3 spring.flyway.user=${PASS}
4 spring.flyway.password=${PASS}
```

Este mecanismo permite mantener la lógica de arranque y las migraciones de base de datos intactas en el código, al tiempo que extrae las credenciales y puertos al fichero de entorno, favoreciendo la seguridad, la portabilidad entre entornos y la facilidad de mantenimiento sin necesidad de recompilar la aplicación para ajustar parámetros del entorno.



### 5.2.5. ¿Cómo conectar con la base de datos PostgreSQL y qué ORM usar?

La conexión con la base de datos PostgreSQL se realiza mediante la configuración del **DataSource** que Spring Boot genera automáticamente a partir de las propiedades definidas en el fichero de configuración.

En concreto, se especifica la URL de conexión, el nombre de usuario y la contraseña, aprovechando la importación de variables de entorno para mantener dichos valores fuera del código fuente.

```
1 # Datos de La base de datos PostgreSQL
2 spring.datasource.url=jdbc:postgresql://${HOST}:${PSQL_PORT}/${PSQL_DBNAME}
3 spring.datasource.username=${PASS}
4 spring.datasource.password=${PASS}
```

Como ORM se ha adoptado la especificación JPA, implícita en Spring Data JPA y normalmente respaldada por Hibernate como proveedor por defecto. Gracias a esta elección, los repositorios se definen como interfaces que extienden `JpaRepository`, y las entidades de dominio se anotan con `@Entity`, `@Table`, `@Id`, etc. Spring Boot, a través de su autoconfiguración, detecta estas clases, crea el `EntityManagerFactory`, un `PlatformTransactionManager` y escanea el paquete de repositorios, de modo que basta con habilitar `@EnableJpaRepositories` para disponer de una capa de persistencia completamente operativa.

### 5.2.6. ¿Cómo conectar con la base de datos Redis y qué ORM usar?

La integración con Redis se lleva a cabo mediante la configuración del cliente Redis de Spring Boot y el uso de un mapeador de objetos específico. En el fichero de propiedades se indican el host, el puerto, las credenciales de acceso, así como el índice de base de.

Adicionalmente, la propiedad `app.redis.bootstrap=true` habilita la inicialización de índices y esquemas en Redis al arrancar la aplicación.

```
1 # Datos de La base de datos Redis
2 spring.data.redis.host=${HOST}
3 spring.data.redis.port=6379
4 spring.data.redis.username=${PASS}
5 spring.data.redis.password=${PASS}
6 spring.data.redis.database=0
7 app.redis.bootstrap=true
```



Para interactuar con Redis como almacen de datos de dominio, se ha optado por la biblioteca Redis OM Spring, cuya dependencia se incorpora al proyecto.

```
1 <!-- Dependencia para Redis -->
2 <dependency>
3   <groupId>com.redis.om</groupId>
4   <artifactId>redis-om-spring</artifactId>
5 </dependency>
```

Este módulo extiende la experiencia de Spring Data con anotaciones ([@RedisHash](#), [@Indexed](#), [@Searchable](#)) que permiten definir entidades de dominio directamente mapeadas a estructuras de Redis y realizar consultas avanzadas por medio de la API de repositorios.

Gracias a la autoconfiguración de Spring Boot, la presencia de redis-om-spring detecta automáticamente las entidades y crea los beans necesarios para el *RedisTemplate*, así como para los repositorios especializados que facilitan operaciones CRUD y búsquedas basadas en índices, ofreciendo un nivel de abstracción y productividad comparable al que JPA proporciona sobre bases de datos relacionales.

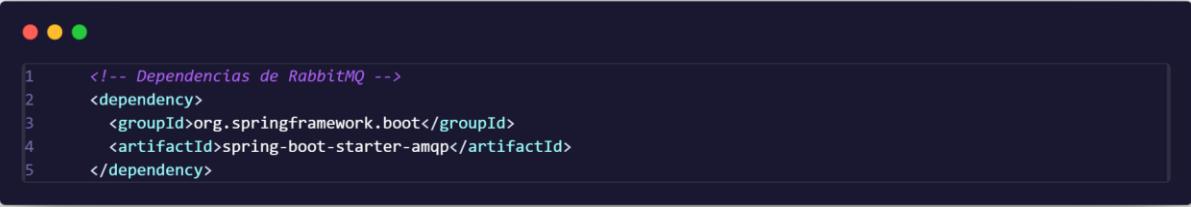
### 5.2.7. ¿Cómo conectar con el gestor de eventos RabbitMQ?

La integración con RabbitMQ se logra mediante la configuración de las propiedades específicas de conexión y la incorporación del starter de Spring AMQP. En el fichero de entorno se definen el host, el puerto y las credenciales de acceso.

```
1 # RabbitMQ
2 spring.rabbitmq.host=${HOST}
3 spring.rabbitmq.port=${RBMQ_PORT}
4 spring.rabbitmq.username=${PASS}
5 spring.rabbitmq.password=${PASS}
```

Al incluir la dependencia `spring-boot-starter-amqp`, Spring Boot proporciona de manera automática un `ConnectionFactory` configurado con dichas propiedades, así como un `RabbitTemplate` para envío de mensajes y un contenedor de listeners para la recepción.





```

1 <!-- Dependencias de RabbitMQ -->
2 <dependency>
3   <groupId>org.springframework.boot</groupId>
4   <artifactId>spring-boot-starter-amqp</artifactId>
5 </dependency>

```

Gracias a esta autoconfiguración, basta con anotar métodos con `@RabbitListener` en los adaptadores de entrada para que la aplicación pueda publicar y suscribirse a colas RabbitMQ sin necesidad de código adicional de bajo nivel, manteniendo así el acoplamiento mínimo y alineándose con el principio de inversión de dependencias.

### 5.2.8. Control de versiones de la base de datos con FlyWay

La gestión de versiones del esquema de base de datos se lleva a cabo mediante Flyway, una herramienta de migraciones que asegura que cada cambio en la estructura (creación de tablas, índices, restricciones, datos iniciales) se aplique de forma ordenada y reproducible en todos los entornos. En el ejemplo que se presenta en la imagen adjunta, la configuración habilita Flyway (`spring.flyway.enabled=true`) y establece las credenciales de conexión (`spring.flyway.user` y `spring.flyway.password`) recuperadas de las variables de entorno, garantizando que las mismas credenciales utilizadas por la aplicación para acceder a la base de datos PostgreSQL sirvan también para las migraciones.

La propiedad `spring.flyway.locations=classpath:db/migrations` indica la carpeta dentro del JAR donde Flyway buscará los scripts numerados que definen cada versión del esquema, siguiendo la convención `V<Versión>_<Descripción>.sql`. De este modo, al iniciar la aplicación, Flyway compara las versiones almacenadas en su tabla de metadatos interna con los archivos presentes en el recurso, detecta las migraciones pendientes y las aplica en orden ascendente.

Para facilitar la adopción en esquemas existentes, se activa el baseline al migrar (`spring.flyway.baseline-on-migrate=true`), con la versión de referencia inicial configurada en 1.0.0 y la descripción “Esquema inicial de GoLearnix”. Esta línea de base permite integrar la gestión de versiones en una base de datos ya poblada sin intentar replicar scripts históricos, marcando automáticamente el punto de partida de las migraciones posteriores.



La validación previa a cada migración (`spring.flyway.validate-on-migrate=true`) comprueba que los scripts ejecutados previamente no hayan sido modificados y que su checksum coincida con el registrado en la tabla de metadatos. En caso de discrepancia, Flyway interrumpe el arranque de la aplicación, evitando posibles incoherencias entre el código y la base de datos.

```
1 # Configuración de Flyway
2 spring.flyway.enabled=true
3 spring.flyway.user=${PASS}
4 spring.flyway.password=${PASS}
5 spring.flyway.locations=classpath:db/migrations
6 # Configuración del Baseline
7 spring.flyway.baseline-on-migrate=true
8 spring.flyway.baseline-version=1.0.0
9 spring.flyway.baselineDescription=Esquema inicial de GoLearnix
10 spring.flyway.validate-on-migrate=true
```

En conjunto, esta configuración garantiza un pipeline de despliegue robusto, donde cada cambio en el modelo relacional se versiona, valida y aplica de forma automática al iniciar el microservicio, simplificando las actualizaciones en entornos de desarrollo, integración continua, staging y producción, y proporcionando un historial completo y auditable de la evolución del esquema de la base de datos.

### 5.2.9. Módulo de iniciación de datos (Patrón de diseño Data Seeding)

El módulo de iniciación de datos, implementado bajo el patrón Data Seeding y contenido en el paquete `init-redis-data`, automatiza la carga inicial y la limpieza de la información en Redis al arrancar la aplicación. Su componente central, la clase `RedisBootstrapper`, anotada con `@ConditionalOnProperty(name = "app.redis.bootstrap", havingValue = "true")` y escuchadora del evento `ApplicationReadyEvent`, orquesta la secuencia de borrado y posterior creación de datos.

```
1 @Util
2 @RequiredArgsConstructor
3 @ConditionalOnProperty(name = "app.redis.bootstrap", havingValue = "true")
4 public class RedisBootstrapper implements ApplicationListener<ApplicationReadyEvent> {
5
6     private final DeleteRedisData deleteRedisData;
7     private final CreateRedisData createRedisData;
8
9     @Override
10    public void onApplicationEvent(ApplicationReadyEvent event) {
11        deleteRedisData.deleteAll();
12        createRedisData.createAll();
13    }
14}
```



Al arrancar, RedisBootstrapper invoca primero el servicio **DeleteRedisData**, encargado de vaciar por completo todas las colecciones de lectura en Redis a través de los distintos repositorios (CategoryReadRepository, CourseReadRepository, etc.). Una vez eliminados los datos anteriores, RedisBootstrapper delega en CreateRedisData la reconstrucción del estado inicial.

```

1  @Util
2  @RequiredArgsConstructor
3  public class DeleteRedisData {
4
5      private final CategoryReadRepository categoryReadRepository;
6      private final CourseReadRepository courseReadRepository;
7      private final EnrollmentReadRepository enrollmentReadRepository;
8      private final LessonReadRepository lessonReadRepository;
9      private final ProgressReadRepository progressReadRepository;
10     private final ReviewReadRepository reviewReadRepository;
11     private final SectionReadRepository sectionReadRepository;
12     private final UserReadRepository userReadRepository;
13
14     @Transactional
15     public void deleteAll() {
16
17         categoryReadRepository.deleteAll();
18         courseReadRepository.deleteAll();
19         enrollmentReadRepository.deleteAll();
20         lessonReadRepository.deleteAll();
21         progressReadRepository.deleteAll();
22         reviewReadRepository.deleteAll();
23         sectionReadRepository.deleteAll();
24         userReadRepository.deleteAll();
25     }
26 }
27
28 }
```

La clase **CreateRedisData** comienza por recuperar el modelo de dominio persistido en PostgreSQL: mediante repositorios JPA (CategoryEntityJpaRepository, CourseEntityJpaRepository, etc.) y mappers (CategoryJpaMapper, CourseJpaMapper, etc.), extrae todas las entidades y las transforma en objetos de dominio. Con un único contenedor **DataHolder**, que agrupa las colecciones de categorías, cursos, inscripciones, lecciones, progresos, reseñas, secciones y usuarios, se dispone de un snapshot completo del estado relacional.

```

1  public record DataHolder(
2      List<Category> categories,
3      List<Course> courses,
4      List<Enrollment> enrollments,
5      List<Lesson> lessons,
6      List<Progress> progresses,
7      List<Review> reviews,
8      List<Section> sections,
9      List<User> users
10 ) {
11
12 }
```

A continuación, CreateRedisData traduce esos objetos de dominio a modelos de lectura específicos de Redis utilizando mappers dedicados (CategoryRedisMapper, CourseRedisMapper, etc.) y persiste cada lista en su correspondiente repositorio de lectura (CategoryReadRepository,



CourseReadRepository, etc.), todo ello dentro de una transacción. Este enfoque garantiza que, tras cada despliegue con `app.redis.bootstrap=true`, Redis quede poblado con datos consistentes y sincronizados con la base de datos relacional, facilitando operaciones de consulta de alta velocidad y reduciendo riesgos de inconsistencia en entornos de caching o sistemas de lectura optimizada.

```

1  @Util
2  @RequiredArgsConstructor
3  public class CreateRedisData {
4
5      private final CategoryEntityJpaRepository categoryEntityJpaRepository;
6      private final CourseEntityJpaRepository courseEntityJpaRepository;
7      private final EnrollmentEntityJpaRepository enrollmentEntityJpaRepository;
8      private final LessonEntityJpaRepository lessonEntityJpaRepository;
9      private final ProgressEntityJpaRepository progressEntityJpaRepository;
10     private final ReviewEntityJpaRepository reviewEntityJpaRepository;
11     private final SectionEntityJpaRepository sectionEntityJpaRepository;
12     private final UserEntityJpaRepository userEntityJpaRepository;
13
14     private final CategoryReadRepository categoryReadRepository;
15     private final CourseReadRepository courseReadRepository;
16     private final EnrollmentReadRepository enrollmentReadRepository;
17     private final LessonReadRepository lessonReadRepository;
18     private final ProgressReadRepository progressReadRepository;
19     private final ReviewReadRepository reviewReadRepository;
20     private final SectionReadRepository sectionReadRepository;
21     private final UserReadRepository userReadRepository;
22
23     private final CategoryJpaMapper categoryJpaMapper;
24     private final CourseJpaMapper courseJpaMapper;
25     private final EnrollmentJpaMapper enrollmentJpaMapper;
26     private final LessonJpaMapper lessonJpaMapper;
27     private final ProgressJpaMapper progressJpaMapper;
28     private final ReviewJpaMapper reviewJpaMapper;
29     private final SectionJpaMapper sectionJpaMapper;
30     private final UserJpaMapper userJpaMapper;
31
32     private final CategoryRedisMapper categoryRedisMapper;
33     private final CourseRedisMapper courseRedisMapper;
34     private final EnrollmentRedisMapper enrollmentRedisMapper;
35     private final LessonRedisMapper lessonRedisMapper;
36     private final ProgressRedisMapper progressRedisMapper;
37     private final ReviewRedisMapper reviewRedisMapper;
38     private final SectionRedisMapper sectionRedisMapper;
39     private final UserRedisMapper userRedisMapper;
40
41     @Transactional
42     public void createAll() {
43         DataHolder data = loadDomainData();
44         saveToRedis(data);
45     }
46
47     private DataHolder loadDomainData() {
48
49         List<Category> categories = categoryEntityJpaRepository.findAll().stream().map(categoryJpaMapper::toDomain).toList();
50         List<Course> courses = courseEntityJpaRepository.findAll().stream().map(courseJpaMapper::toDomain).toList();
51         List<Enrollment> enrollments = enrollmentEntityJpaRepository.findAll().stream().map(enrollmentJpaMapper::toDomain).toList();
52         List<Lesson> lessons = lessonEntityJpaRepository.findAll().stream().map(lessonJpaMapper::toDomain).toList();
53         List<Progress> progresses = progressEntityJpaRepository.findAll().stream().map(progressJpaMapper::toDomain).toList();
54         List<Review> reviews = reviewEntityJpaRepository.findAll().stream().map(reviewJpaMapper::toDomain).toList();
55         List<Section> sections = sectionEntityJpaRepository.findAll().stream().map(sectionJpaMapper::toDomain).toList();
56         List<User> users = userEntityJpaRepository.findAll().stream().map(userJpaMapper::toDomain).toList();
57
58         return new DataHolder(categories, courses, enrollments, lessons, progresses, reviews, sections, users);
59     }
60
61     private void saveToRedis(DataHolder data) {
62
63         categoryReadRepository.saveAll(data.categories().stream().map(categoryRedisMapper::toReadModel).collect(Collectors.toList()));
64         courseReadRepository.saveAll(data.courses().stream().map(courseRedisMapper::toReadModel).collect(Collectors.toList()));
65         enrollmentReadRepository.saveAll(data.enrollments().stream().map(enrollmentRedisMapper::toReadModel).collect(Collectors.toList()));
66         lessonReadRepository.saveAll(data.lessons().stream().map(lessonRedisMapper::toReadModel).collect(Collectors.toList()));
67         progressReadRepository.saveAll(data.progresses().stream().map(progressRedisMapper::toReadModel).collect(Collectors.toList()));
68         reviewReadRepository.saveAll(data.reviews().stream().map(reviewRedisMapper::toReadModel).collect(Collectors.toList()));
69         sectionReadRepository.saveAll(data.sections().stream().map(sectionRedisMapper::toReadModel).collect(Collectors.toList()));
70         userReadRepository.saveAll(data.users().stream().map(userRedisMapper::toReadModel).collect(Collectors.toList()));
71
72     }
73 }
74
75 }
```

### 5.2.10. Patrón saga

La implementación del patrón Saga en este proyecto tiene como propósito garantizar la consistencia eventual entre las bases de datos PostgreSQL (como fuente de verdad) y Redis (como almacenamiento optimizado para consultas). Este patrón resulta especialmente útil en arquitecturas distribuidas o en sistemas que manejan múltiples fuentes de datos, permitiendo manejar fallos parciales mediante acciones compensatorias.

En el contexto de GoLearnix, se ha utilizado un enfoque basado en eventos internos de Spring para implementar este patrón. Las operaciones críticas, como creación, actualización o eliminación de entidades generan eventos específicos. Estos eventos son escuchados por componentes dedicados que replican los cambios en Redis, y en caso de fallo, aplican una operación compensatoria en PostgreSQL para mantener la coherencia.

Esto ofrece varias ventajas como:

- **Resiliencia:** Permite manejar errores en Redis sin comprometer la integridad del sistema.
- **Desacoplamiento:** La lógica de negocio no se ve afectada por la infraestructura de almacenamiento en caché.
- **Mantenibilidad:** La gestión de eventos y acciones compensatorias está claramente delimitada por capas y responsabilidades.

#### 5.2.10.1. Publicadores de eventos

Los publicadores de eventos (**UserEventPublisher**, **CourseEventPublisher**) son clases que encapsulan la lógica de disparar eventos de dominio. Estos eventos son instancias de clases como **UserDeletedEvent**, **CourseCreatedEvent**, **CourseUpdatedEvent**, etc. Cada una representa un cambio significativo en el estado de una entidad.

Por ejemplo, al eliminar un usuario en el dominio, se ejecuta: `publisher.publishUserDeleted(oldUser);`

Esto lanza un **UserDeletedEvent** al contexto de la aplicación, activando a los listeners correspondientes.



```

1 @Publisher
2 @RequiredArgsConstructor
3 public class UserEventPublisher {
4
5     private final ApplicationEventPublisher publisher;
6
7     public void publishUserDeleted(User oldUser) {
8         publisher.publishEvent(new UserDeletedEvent(oldUser));
9     }
10 }
11
12

```

```

1 @Publisher
2 @RequiredArgsConstructor
3 public class CourseEventPublisher {
4
5     private final ApplicationEventPublisher publisher;
6
7     public void publishCourseCreated(Course course) {
8         publisher.publishEvent(new CourseCreatedEvent(course));
9     }
10
11     public void publishCourseUpdated(Course oldCourse, Course newCourse) {
12         publisher.publishEvent(new CourseUpdatedEvent(oldCourse, newCourse));
13     }
14
15     public void publishCourseDeleted(Course oldCourse) {
16         publisher.publishEvent(new CourseDeletedEvent(oldCourse));
17     }
18 }
19

```

### 5.2.10.2. Eventos de dominio

Los eventos (**CourseCreatedEvent**, **CourseUpdatedEvent**, etc.) actúan como contenedores inmutables que transportan los datos necesarios para replicar o compensar una acción. Su uso facilita el desacoplamiento entre la lógica de negocio y la lógica de persistencia en Redis, permitiendo extender el comportamiento sin modificar la fuente.

```

1 public record CourseCreatedEvent(Course course) {
2
3 }

```

```

1 public record CourseUpdatedEvent(Course oldCourse, Course newCourse) {
2
3 }

```

```

1 public record CourseDeletedEvent(Course oldCourse) {
2
3 }

```

```

1 public record UserDeletedEvent(User oldUser) {
2
3 }
4

```



### 5.2.10.3. Listeners de eventos y compensaciones

Las clases CourseEventListener y UserEventListener representan los sagas coordinadores, encargados de realizar las acciones en Redis y, si estas fallan, ejecutar las acciones compensatorias en PostgreSQL.

Ejemplo de comportamiento:

#### 1. Creación de curso (onCourseCreated):

- Se intenta guardar el nuevo curso en Redis.
- Si Redis falla, se elimina la entidad creada en PostgreSQL para evitar inconsistencias.

```

1  @EventListener
2  public void onCourseCreated(CourseCreatedEvent event) {
3
4      Course course = event.course();
5
6      try {
7          courseQueryRepositoryPort.save(course);
8      } catch (Exception redisEx) {
9          courseCommandRepositoryPort.delete(course.getId());
10     }
11 }
12 }
```

#### 2. Actualización de curso (onCourseUpdated):

- Se intenta actualizar el curso en Redis.
- Si falla, se revierte el cambio en PostgreSQL restaurando el estado anterior (oldCourse).

```

1  @EventListener
2  public void onCourseUpdated(CourseUpdatedEvent event) {
3
4      Course oldCourse = event.oldCourse();
5      Course newCourse = event.newCourse();
6
7      try {
8          courseQueryRepositoryPort.update(newCourse);
9      } catch (Exception redisEx) {
10          courseCommandRepositoryPort.save(oldCourse);
11     }
12 }
```

#### 3. Eliminación de curso o usuario (onCourseDeleted, onUserDeleted):

- Se elimina la entidad de Redis.
- Si la operación falla, se reintroduce la entidad en PostgreSQL para preservar su disponibilidad.



```

1  @EventListener
2  public void onCourseDeleted(CourseDeletedEvent event) {
3
4      Course oldCourse = event.oldCourse();
5
6      try {
7          courseQueryRepositoryPort.delete(oldCourse.getId());
8      } catch (Exception redisEx) {
9          courseCommandRepositoryPort.save(oldCourse);
10     }
11 }
12 }
```

```

1  @EventListener
2  public void onUserDeleted(UserDeletedEvent event) {
3
4      User oldUser = event.oldUser();
5
6      try {
7          userQueryRepositoryPort.delete(oldUser.getId());
8      } catch (Exception redisEx) {
9          userCommandRepositoryPort.save(oldUser);
10     }
11 }
12 }
```

### 5.2.11. Patrón repository

Se aplica el Patrón Repository para abstraer el acceso a datos y aislar la lógica de persistencia del dominio. Un repositorio actúa como intermediario entre el dominio y la base de datos, encapsulando varios DAOs cuando es necesario.

Aunque un repositorio puede coordinar múltiples DAOs, para simplificar, cada servicio del dominio interactúa directamente con varios repositorios, definidos como puertos de salida (RepositoryPort) según la arquitectura hexagonal. Esto facilita el desacoplamiento entre el dominio y la infraestructura.

Este enfoque mejora la mantenibilidad, permite cambiar tecnologías de persistencia sin afectar al dominio, y hace que la lógica de negocio sea más clara y testable.

```

1  @DomainService
2  @RequiredArgsConstructor
3  public class CourseServiceUseCase implements CourseServicePort {
4
5      private final CourseCommandRepositoryPort courseCommandRepositoryPort;
6      private final CourseQueryRepositoryPort courseQueryRepositoryPort;
```



### 5.2.12. Patrón assembler

Para las operaciones de inserción y actualización de cursos se ha empleado el Patrón Assembler, que centraliza la lógica de transformación y resolución de dependencias entre objetos de dominio.

```

1  @Override
2  @Transactional
3  public void create(Course course) {
4      Course newCourse = courseAssembler.assemble(course);
5      courseCommandRepositoryPort.save(newCourse);
6
7      eventPublisher.publishCourseCreated(newCourse);
8  }
9
10 @Override
11 @Transactional
12 public void update(Integer id, Course course) {
13     Course oldCourse = getById(id);
14     Course newCourse = courseAssembler.assemble(oldCourse, course);
15
16     courseCommandRepositoryPort.save(newCourse);
17     eventPublisher.publishCourseUpdated(oldCourse, newCourse);
18 }
```

La clase **CourseAssembler**, anotada como `@Assembler`, recibe dos puertos de servicio, uno para usuarios y otro para categorías, con los que resuelve únicamente las relaciones simples del curso: el instructor y la categoría.

Durante el ensamblaje, los campos primarios (título y descripción) se copian directamente del objeto fuente al destino, mientras que los identificadores de instructor y de categoría se transforman en entidades completas mediante llamadas a `userServicePort.getById(...)` y `categoryServicePort.getById(...)`.

```

1  @Assembler
2  @RequiredArgsConstructor
3  public class CourseAssembler {
4
5      private final UserServicePort userServicePort;
6      private final CategoryServicePort categoryServicePort;
7
8
9      public Course assemble(Course course) {
10         return assemble(course, course);
11     }
12
13     public Course assemble(Course target, Course source) {
14         target.setTitle(source.getTitle());
15         target.setDescription(source.getDescription());
16         target.setInstructor(resolveInstructor(source));
17         target.setCategory(resolveCategory(source));
18
19         return target;
20     }
21
22     private User resolveInstructor(Course course) {
23         return userServicePort.getById(course.getInstructor().getId());
24     }
25
26     private Category resolveCategory(Course course) {
27         return categoryServicePort.getById(course.getCategory().getId());
28     }
29
30 }
```



Gracias a este enfoque, la lógica de mapeo queda aislada en un componente especializado, y se respeta el principio de responsabilidad única al delegar en servicios externos la obtención de datos relacionados, dejando para su propio módulo el tratamiento de relaciones más complejas.

### 5.2.13. Consumidor de eventos para RabbitMQ

El consumidor de eventos para RabbitMQ se compone de cuatro bloques principales, cada uno especializado en una responsabilidad concreta dentro del flujo de procesamiento.

En primer lugar, el adaptador **UserRabbitConsumerAdapter** escucha la cola `user.deleted.queue`. Al recibir un mensaje, el método `delete` deserializa automáticamente el payload en un objeto `UserDeletedEvent`, invoca el servicio de dominio `userServicePort.delete(...)` utilizando el identificador de usuario extraído del mensaje y, finalmente, confirma manualmente el procesamiento del mensaje con `channel.basicAck(tag, false)`. La anotación `@RabbitListener` junto con la referencia a `rabbitListenerContainerFactory` garantiza que el adaptador utilice el contenedor configurado para el consumo, con modo de reconocimiento manual y políticas de reintento.

```

1  @EventAdapter
2  @RequiredArgsConstructor
3  public class UserRabbitConsumerAdapter {
4
5      private final UserServicePort userServicePort;
6
7      @RabbitListener(queues = "user.deleted.queue", containerFactory = "rabbitListenerContainerFactory")
8      public void delete(UserDeletedEvent event, Channel channel, @Header(AmqpHeaders.DELIVERY_TAG) long tag) throws IOException {
9
10         userServicePort.delete(UUID.fromString(event.getData().getUserId()));
11         channel.basicAck(tag, false);
12     }
13 }
14
15 }
```

El DTO **UserDeletedEvent** define la estructura del mensaje recibido: un campo `event` que identifica el tipo de evento y un objeto anidado `data` de tipo **UserDetail**. Este último contiene el identificador del usuario (`userId`) y su correo electrónico, mapeados desde propiedades en `snake_case` gracias a la configuración global del **ObjectMapper**. Ambas clases están anotadas con validaciones (`@NotBlank`, `@Valid`) para asegurar la integridad de los datos antes de su uso.



```

1 @Data
2 @NoArgsConstructor
3 @AllArgsConstructor
4 public class UserDeletedEvent {
5
6     @NotBlank
7     private String event;
8
9     @Valid
10    private UserDataDetail data;
11 }
12 
```

```

1 @Data
2 @NoArgsConstructor
3 @AllArgsConstructor
4 @JsonNaming(PropertyNamingStrategies.SnakeCaseStrategy.class)
5 public class UserDataDetail {
6
7     private String userID;
8     private String email;
9
10 } 
```

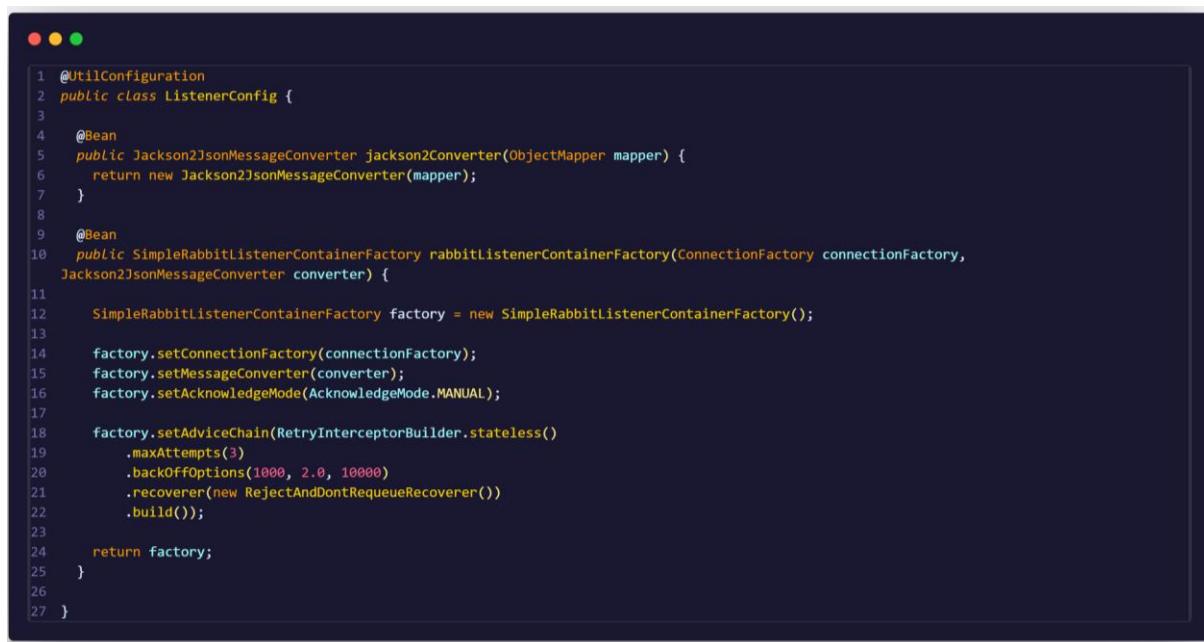
La clase **JacksonConfig** provee un *ObjectMapper* central configurado con la estrategia SNAKE\_CASE y la aceptación de propiedades insensibles a mayúsculas, lo que unifica el tratamiento de nombres de campos en todos los conversores JSON de la aplicación.

```

1 @UtilConfiguration
2 public class JacksonConfig {
3
4     @Bean
5     public ObjectMapper objectMapper() {
6         return JsonMapper.builder()
7             .propertyNamingStrategy(PropertyNamingStrategies.SNAKE_CASE)
8             .enable(com.fasterxml.jackson.databind.MapperFeature.ACCEPT_CASE_INSENSITIVE_PROPERTIES)
9             .build();
10    }
11 }
12 
```

Por último, **ListenerConfig** define el *SimpleRabbitListenerContainerFactory* utilizado por el adaptador. Se inyecta un *ConnectionFactory* y un *Jackson2JsonMessageConverter* basado en el *ObjectMapper* anterior. El contenedor se establece en modo de reconocimiento manual ([AcknowledgeMode.MANUAL](#)) para controlar explícitamente los ACK/NACK, y se añade una cadena de consejos de reintento que reintentará hasta tres veces con back-off exponencial, rechazando el mensaje y evitando su reencolado en caso de fallo definitivo.





```

1  @UtilConfiguration
2  public class ListenerConfig {
3
4      @Bean
5      public Jackson2JsonMessageConverter jackson2Converter(ObjectMapper mapper) {
6          return new Jackson2JsonMessageConverter(mapper);
7      }
8
9      @Bean
10     public SimpleRabbitListenerContainerFactory rabbitListenerContainerFactory(ConnectionFactory connectionFactory,
11         Jackson2JsonMessageConverter converter) {
12
13         SimpleRabbitListenerContainerFactory factory = new SimpleRabbitListenerContainerFactory();
14
15         factory.setConnectionFactory(connectionFactory);
16         factory.setMessageConverter(converter);
17         factory.setAcknowledgeMode(AcknowledgeMode.MANUAL);
18
19         factory.setAdviceChain(RetryInterceptorBuilder.stateless()
20             .maxAttempts(3)
21             .backOffOptions(1000, 2.0, 10000)
22             .recoverer(new RejectAndDontRequeueRecoverer())
23             .build());
24
25         return factory;
26     }
27 }

```

De este modo, se consigue un procesamiento de mensajes robusto, tolerante a errores y alineado con las buenas prácticas de mensajería asíncrona.

#### 5.2.14. Control de errores en RabbitMQ

El control de errores en los consumidores de RabbitMQ se centraliza en la clase **RabbitErrorHandler**, que implementa la interfaz **RabbitListenerErrorHandler** para interceptar cualquier excepción que se produzca durante el procesamiento de un mensaje. Esta solución garantiza un manejo uniforme de los fallos y la instrumentación automática de métricas y registros de auditoría.

En el método `handleError`, en primer lugar, se extraen de la instancia de Message de AMQP los metadatos relevantes: la cola que estaba consumiendo el mensaje (`consumerQueue`), la clave de enrutamiento (`receivedRoutingKey`) y los encabezados originales. A continuación, se determina el motivo del error, intentando obtener el mensaje de la causa raíz (`exception.getCause().getMessage()`), o recurriendo al mensaje de la excepción principal si no existe causa definida.

Para medir la incidencia de errores, se crea o recupera un contador en el `MeterRegistry` bajo el nombre `rabbitmq.listener.errors`. A este contador se le asocian dos etiquetas (tags): el nombre de la cola y el tipo de excepción que ha provocado el fallo. De este modo, se puede recoger métricas por cola y por tipo de error, permitiendo alertas y análisis detallados de la fiabilidad del consumidor.



Simultáneamente, se registra un mensaje de error en el logger, con nivel ERROR, que incluye la cola, la clave de enrutamiento, los encabezados y la descripción del fallo, junto con la traza de la excepción raíz. Esta práctica facilita la investigación de incidentes y la localización de la causa en los sistemas de logging.

Por último, para indicar a RabbitMQ que el mensaje no debe reintentarse ni reencolarse, se lanza una excepción ***AmqpRejectAndDontRequeueException***.

Esta excepción específica interrumpe el flujo de reintentos y evita que el broker vuelva a entregar el mensaje, garantizando que los mensajes erróneos no bloquen la cola ni generen bucles infinitos de procesamiento fallido.

```

● ● ●
1 @Slf4j
2 @Util
3 @RequiredArgsConstructor
4 public class RabbitErrorHandler implements RabbitListenerErrorHandler {
5
6     private final MeterRegistry meterRegistry;
7
8     @Override
9     public Object handleError(Message amqpMessage, Channel channel, org.springframework.messaging.Message<?> springMsg,
10                           ListenerExecutionFailedException exception) {
11
12         String queue      = amqpMessage.getMessageProperties().getConsumerQueue();
13         String routingKey = amqpMessage.getMessageProperties().getReceivedRoutingKey();
14         Object headers   = amqpMessage.getMessageProperties().getHeaders();
15         String reason    = exception.getCause() != null
16             ? exception.getCause().getMessage()
17             : exception.getMessage();
18
19         Counter.builder("rabbitmq.listener.errors")
20             .description("Número de errores en RabbitMQ listener")
21             .tag("queue", queue)
22             .tag("exception", exception.getClass().getSimpleName())
23             .register(meterRegistry)
24             .increment();
25
26         log.error("Error en listener [queue={}, routingKey={}, headers={}]: {}", queue, routingKey, headers, reason, exception.getCause());
27
28         throw new AmqpRejectAndDontRequeueException(reason, exception);
29     }
30
31 }
```

### 5.2.15. Control de errores en REST

En la capa REST, la gestión de errores se centraliza mediante un objeto de respuesta común y un controlador de asesoramiento global. La clase **ErrorMessage** define la estructura de la respuesta de error, incluyendo el tipo de excepción, el mensaje descriptivo y, cuando procede, una lista de errores de validación. Su diseño *utiliza `@JsonInclude(JsonInclude.Include.NON_NULL)`* para omitir del JSON cualquier campo nulo, de modo que las respuestas sean más limpias y concisas.



```

1  @Getter
2  @ToString
3  @JsonInclude(JsonInclude.Include.NON_NULL)
4  public class ErrorMessage {
5
6      private final String error;
7      private final String message;
8      private final List<String> validationErrors;
9
10     public ErrorMessage(Exception exception) {
11         this.error = exception.getClass().getSimpleName();
12         this.message = exception.getMessage();
13         this.validationErrors = null;
14     }
15
16     public ErrorMessage(List<String> validationErrors) {
17         this.error = "ValidationException";
18         this.message = "VALIDATION EXCEPTION";
19         this.validationErrors = validationErrors.isEmpty() ? null : validationErrors;
20     }
21 }

```

Por su parte, **ApiControllerAdvice**, anotado con `@ControllerAdvice`, intercepta las excepciones lanzadas por los controladores y las traduce en respuestas HTTP adecuadas. Cada método de manejo se decora con `@ExceptionHandler` para el tipo de excepción correspondiente, `@ResponseStatus` para asignar el código HTTP correcto y `@ResponseBody` para serializar automáticamente un `ErrorMessage`.

Así, un `ResourceNotFoundException` produce un 404 Not Found, un `MethodArgumentNotValidException` (violación de restricciones Bean Validation) genera un 400 Bad Request con la lista detallada de campos inválidos, y cualquier excepción sin manejo específico devuelve un 500 Internal Server Error. Los errores de autorización (`AccessDeniedException` o `AuthorizationDeniedException`) reciben un 403 Forbidden, mientras que la excepción de negocio `UserAlreadyEnrolledInCourse` se traduce también en un 400 Bad Request.



```

1  @ControllerAdvice
2  public class ApiControllerAdvice {
3
4      @ResponseStatus(HttpStatus.NOT_FOUND)
5      @ExceptionHandler(ResourceNotFoundException.class)
6      @ResponseBody
7      public ErrorMessage notFoundRequest(ResourceNotFoundException exception) {
8          return new ErrorMessage(exception);
9      }
10
11     @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
12     @ExceptionHandler(Exception.class)
13     @ResponseBody
14     public ErrorMessage handleGeneralException(Exception exception) {
15         return new ErrorMessage(exception);
16     }
17
18     @ResponseStatus(HttpStatus.BAD_REQUEST)
19     @ExceptionHandler(MethodArgumentNotValidException.class)
20     @ResponseBody
21     public ErrorMessage handleValidationException(MethodArgumentNotValidException exception) {
22
23         List<String> validationErrors = exception.getBindingResult().getFieldErrors().stream()
24             .map(fieldError -> String.format("Campo '%s': %s", fieldError.getField(),
25                 fieldError.getDefaultMessage()))
26             .collect(Collectors.toList());
27
28         return new ErrorMessage(validationErrors);
29     }
30
31
32     @ResponseStatus(HttpStatus.BAD_REQUEST)
33     @ExceptionHandler(UserAlreadyEnrolledInCourse.class)
34     @ResponseBody
35     public ErrorMessage handleUserAlreadyEnrolledInCourseException(UserAlreadyEnrolledInCourse exception) {
36         return new ErrorMessage(exception);
37     }
38
39     @ResponseStatus(HttpStatus.FORBIDDEN)
40     @ExceptionHandler({AuthorizationDeniedException.class, AccessDeniedException.class})
41     @ResponseBody
42     public ErrorMessage handleAuthorizationDeniedException(AuthorizationDeniedException exception) {
43         return new ErrorMessage(exception);
44     }
45
46 }

```

Este enfoque unificado garantiza que todos los errores se comuniquen de forma consistente, con payloads estructurados y códigos de estado alineados con las convenciones REST, facilitando tanto el consumo programático como la depuración y monitorización.

### 5.2.16. Configuración de roles y seguridad en REST

La configuración de seguridad REST se basa en JWT y en un esquema de roles jerárquicos, de modo que cada petición queda autenticada y autorizada de forma declarativa. Para interpretar el token, se define un bean de **JwtDecoder** que decodifica el secreto en hexadecimal y construye un decodificador Nimbus con algoritmo HS256; a su vez, un **JwtRoleConverter** traduce el claim role en autoridades Spring Security de la forma *ROLE\_{ROL}*.



```

1  @UtilConfiguration
2  public class JwtDecoderConfig {
3
4      @Bean
5      public JwtDecoder jwtDecoder(@Value("${jwt.secret}") String hexSecret) throws DecoderException {
6
7          byte[] keyBytes = Hex.decodeHex(hexSecret.toCharArray());
8          SecretKeySpec secretKey = new SecretKeySpec(keyBytes, "HmacSHA256");
9
10         return NimbusJwtDecoder.withSecretKey(secretKey).macAlgorithm(MacAlgorithm.HS256).build();
11     }
12 }
13 }
```

```

1  @Util
2  public class JwtRoleConverter implements Converter<Jwt, Collection<GrantedAuthority>> {
3
4      private static final String ROLE_CLAIM = "role";
5
6      @Override
7      public Collection<GrantedAuthority> convert(Jwt jwt) {
8
9          String role = jwt.getClaimAsString(ROLE_CLAIM);
10         return List.of(new SimpleGrantedAuthority("ROLE_" + role.toUpperCase()));
11     }
12 }
13 }
14 }
```

El componente **JwtUtil**, respaldado por el **SecurityContextHolder**, extrae el identificador de usuario (sub) y el rol directamente del JWT, y el **CurrentUserHelper** lo expone mediante métodos `getId()` y `getRole()` que los servicios pueden invocar sin lidiar con detalles de seguridad.

```

1  @Util
2  public class JwtUtil {
3
4      private Jwt getJwt() {
5          return (Jwt) SecurityContextHolder
6              .getContext()
7              .getAuthentication()
8              .getPrincipal();
9      }
10
11     public UUID getUserId() {
12         String sub = getJwt().getSubject();
13         return UUID.fromString(sub);
14     }
15
16     public String getRole() {
17         return getJwt().getClaimAsString("role");
18     }
19
20 }
21 }
```



La jerarquía de roles se configura a través de un bean **RoleHierarchy** que establece que *ROLE\_ADMIN* supera a *ROLE\_INSTRUCTOR*, y este a su vez a *ROLE\_STUDENT*, habilitando herencia de permisos en las expresiones de método.

```
1 @UtilConfiguration
2 public class RoleHierarchyConfig {
3
4     @Bean
5     public RoleHierarchy roleHierarchy() {
6
7         String hierarchyDefinition = String.join("\n",
8             UserRole.ROLE_ADMIN.name()      + " > " + UserRole.ROLE_INSTRUCTOR.name(),
9             UserRole.ROLE_INSTRUCTOR.name() + " > " + UserRole.ROLE_STUDENT.name()
10        );
11
12        return RoleHierarchyImpl.fromHierarchy(hierarchyDefinition);
13    }
14
15    @Bean
16    public MethodSecurityExpressionHandler methodSecurityExpressionHandler(RoleHierarchy
17        roleHierarchy) {
18
19        DefaultMethodSecurityExpressionHandler handler = new DefaultMethodSecurityExpressionHandler
20        ();
21        handler.setRoleHierarchy(roleHierarchy);
22
23        return handler;
24    }
}
```

El **SecurityFilterChain** deshabilita CSRF y permite libre acceso a las rutas GET de consulta de cursos, requiriendo autenticación para el resto. En su **oauth2ResourceServer** se integra el decodificador y el convertidor de roles, de forma que todos los filtros posteriores cuentan con un Authentication correctamente poblado.



```

1  @UtilConfiguration
2  @EnableMethodSecurity
3  public class SecurityConfig {
4
5      @Value("${api.base-path}")
6      private String basePath;
7
8      private final JwtDecoder jwtDecoder;
9      private final JwtAuthenticationConverter jwtConverter;
10
11     public SecurityConfig(JwtDecoder jwtDecoder, JwtRoleConverter roleConverter) {
12         this.jwtDecoder = jwtDecoder;
13         this.jwtConverter = new JwtAuthenticationConverter();
14         this.jwtConverter.setJwtGrantedAuthoritiesConverter(roleConverter);
15     }
16
17     @Bean
18     public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
19         http
20             .csrf(AbstractHttpConfigurer::disable)
21             .authorizeHttpRequests(auth -> auth
22                 .requestMatchers(HttpServletRequest.GET,
23                     basePath + "/courses",
24                     basePath + "/courses/*"
25                 ).permitAll()
26                 .anyRequest().authenticated()
27             )
28             .oauth2ResourceServer(oauth2 -> oauth2
29                 .jwt(jwt -> jwt
30                     .decoder(jwtDecoder)
31                     .jwtAuthenticationConverter(jwtConverter)
32                 )
33             );
34
35         return http.build();
36     }
37
38 }

```

Finalmente, una clase de datos de utilidad (**UserRoleData**) expone los nombres de los roles como constantes, facilitando su uso en expresiones `@PreAuthorize("hasRole(@role.INSTRUCTOR)")` y asegurando consistencia de valores en toda la aplicación.

```

1  @Util("role")
2  public class UserRoleData {
3
4      public final String ADMIN      = UserRole.ROLE_ADMIN.name();
5      public final String INSTRUCTOR = UserRole.ROLE_INSTRUCTOR.name();
6      public final String STUDENT   = UserRole.ROLE_STUDENT.name();
7
8  }
9

```



### 5.2.17. Funcionamiento de la aplicación

El flujo de ejecución de cada operación se articula también en torno a una arquitectura hexagonal de capas bien diferenciadas, adaptadores de entrada, aplicación (servicios / casos de uso y dominio) e infraestructura, y hace uso de filtros de seguridad para autorizar las peticiones.

Cuando un cliente emite una llamada HTTP, el controlador correspondiente (adaptador REST) la recibe, deserializa los parámetros y valida los permisos mediante las anotaciones `@PreAuthorize`. A continuación, el controlador delega en el servicio de aplicación (implementación del puerto de entrada) transmitiendo un objeto de dominio.

Dentro de la capa de aplicación, el caso de uso / servicio orquesta la lógica de negocio: invoca al ensamblador (Assembler) para fusionar datos de entrada y estado previo, aplica las reglas de dominio y, si procede, publica eventos de dominio (`CourseCreatedEvent`, `UserDeletedEvent`, etc.) para notificar cambios a otros subsistemas. Asimismo, este servicio actúa sobre los repositorios de salida (puertos) para persistir o recuperar entidades, aprovechando Spring Data JPA para PostgreSQL y Redis OM Spring para Redis.

En la capa de infraestructura, los adaptadores de repositorio (implementaciones JPA o Redis) traducen las entidades de dominio a modelos de persistencia y ejecutan las operaciones de base de datos dentro de transacciones gestionadas por Spring. Paralelamente, los adaptadores de mensajería emplean `@RabbitListener` para integrar RabbitMQ, garantizando el desacoplo entre la lógica central y las tareas asíncronas.

Finalmente, el controlador construye una respuesta uniforme, encapsulada en un `ResponseType` con el código HTTP adecuado y un cuerpo JSON estandarizado, que puede incluir datos, mensajes de éxito o estructuras de error. De este modo, cada petición cierra el ciclo de la misma manera que en Go, pero integrando las ventajas del ecosistema Spring Boot: autoconfiguración, declaratividad de seguridad, gestión de transacciones y un sistema de eventos interno que alimenta las operaciones de replicación y compensación sin acoplar el dominio a detalles de infraestructura.

A continuación, se presentan, apoyados en diagramas, los flujos de invocación entre métodos para cada caso de uso.



### 5.2.17.1. Ver todos los cursos

El controlador **GET /courses** llama a **courseServicePort.getAll()**. El caso de uso consulta el repositorio de lectura (**CourseQueryRepositoryPort**), que en Redis o PostgreSQL devuelve una lista de proyecciones **CourseGetAllProjection**. El controlador envuelve ese listado en un **ResponseEntity.ok(...)** y lo retorna al cliente.

#### Controlador adaptador de entrada → Servicio

```
1 @GetMapping
2 public ResponseEntity<List<CourseGetAllProjection>> getAll() {
3     return ResponseEntity.ok(courseServicePort.getAll());
4 }
```

#### Servicio → Repositorio puerto de lectura

```
1 @Override
2 public List<CourseGetAllProjection> getAll() {
3     return courseQueryRepositoryPort.getAll();
4 }
```

#### Repositorio adaptador de salida de lectura → Repositorio del ORM

```
1 @Override
2 public List<CourseGetAllProjection> getAll() {
3     return courseReadRepository.findAllBy();
4 }
```



### Repositorio de lectura del ORM → Base de datos

```
1 public interface CourseReadRepository extends RedisDocumentRepository<CourseReadModel, Integer> {
2
3     List<CourseGetAllProjection> findAllBy();
4
5 }
```

**NOTA:** Se usa una proyección para devolver solo los campos necesarios y mejorar el rendimiento en la consulta a la base de datos al solo traer los campos requeridos.

#### 5.2.17.2. Ver los detalles de un curso

Al invocar **GET /courses/{id}**, el controlador extrae el id y llama a **courseServicePort.getById(id)**. El servicio consulta el repositorio de consulta (**CourseQueryRepositoryPort**), recupera la entidad completa **Course** o lanza **ResourceNotFoundException** si no existe, y la devuelve en un **ResponseEntity.ok(course)**.

### Controlador adaptador de entrada → Servicio

```
1 @GetMapping("/{id}")
2 public ResponseEntity<Course> getById(@PathVariable Integer id) {
3     return ResponseEntity.ok(courseServicePort.getById(id));
4 }
```

### Servicio → Repositorio puerto de lectura

```
1 @Override
2 public Course getById(Integer id) throws ResourceNotFoundException {
3     return courseQueryRepositoryPort.getById(id)
4         .orElseThrow(() -> new ResourceNotFoundException("Course not found with id: " + id));
5 }
```



Repositorio adaptador de salida de lectura → Repositorio del ORM

```

1  @Override
2  public Optional<Course> getById(Integer id) {
3      return courseReadRepository.findById(id)
4          .map(courseRedisMapper::toDomain);
5  }

```

Repositorio de lectura del ORM → Base de datos

```

1  public interface CourseReadRepository extends RedisDocumentRepository<CourseReadModel, Integer> {
2
3      List<Course GetAllProjection> findAllBy();
4
5  }

```

**5.2.17.3. Insertar un curso**

El controlador **POST /courses**, protegido con `@PreAuthorize("hasRole(@role.INSTRUCTOR)")`, recibe el JSON, lo valida y crea un objeto **Course**. A continuación invoca `courseServicePort.create(course)`. Dentro del servicio, el **CourseAssembler** resuelve relaciones simples (instructor y categoría), luego el puerto de comando (**CourseCommandRepositoryPort**) persiste la entidad en PostgreSQL y publica un **CourseCreatedEvent** para replicar en Redis. Finalmente, el controlador devuelve `ResponseEntity.ok().build()`.

Controlador adaptador de entrada → Servicio

```

1  @PostMapping
2  @PreAuthorize("hasRole(@role.INSTRUCTOR)")
3  public ResponseEntity<Void> create(@RequestBody @Valid Course course) {
4      courseServicePort.create(course);
5      return ResponseEntity.ok().build();
6  }
7

```



Servicio → Repositorio puerto de lectura

```

1  @Override
2  @Transactional
3  public void create(Course course) {
4      Course newCourse = courseAssembler.assemble(course);
5      courseCommandRepositoryPort.save(newCourse);
6
7      eventPublisher.publishCourseCreated(newCourse);
8  }

```

Repositorio adaptador de salida de escritura → Repositorio del ORM

```

1  @Override
2  public void save(Course course) {
3      courseEntityJpaRepository.save(courseJpaMapper.toJpaEntity(course));
4  }

```

Repositorio de escritura del ORM → Base de datos

```

1  public interface CourseEntityJpaRepository extends JpaRepository<CourseEntity, Integer> {
2
3  }

```

**NOTA:** El flujo de los eventos está explicado en la sección [5.2.10. Patrón saga](#).

**5.2.17.4. Actualizar un curso**

La petición **PUT /courses/{id}** pasa por el mismo filtro de seguridad. El controlador extrae id y cuerpo, crea dos instancias de **Course** (target y source) y llama a **courseServicePort.update(id, source)**. El servicio recupera la entidad existente, la pasa al **CourseAssembler** junto con los nuevos datos, la



actualiza vía **CourseCommandRepositoryPort.update(...)**, emite un **CourseUpdatedEvent** y responde con **ResponseEntity.ok()**.

#### Controlador adaptador de entrada → Servicio

```
● ● ●
1 @PutMapping("/{id}")
2 @PreAuthorize("hasRole(@role.INSTRUCTOR)")
3 public ResponseEntity<Void> update(@PathVariable Integer id, @RequestBody @Valid Course course)
{
4     courseServicePort.update(id, course);
5     return ResponseEntity.ok().build();
6 }
```

#### Servicio → Repositorio puerto de lectura

```
● ● ●
1 @Override
2 @Transactional
3 public void update(Integer id, Course course) {
4     Course oldCourse = getById(id);
5     Course newCourse = courseAssembler.assemble(oldCourse, course);
6
7     courseCommandRepositoryPort.save(newCourse);
8     eventPublisher.publishCourseUpdated(oldCourse, newCourse);
9 }
```

#### Repositorio adaptador de salida de escritura → Repositorio del ORM

```
● ● ●
1 @Override
2 public void save(Course course) {
3     courseEntityJpaRepository.save(courseJpaMapper.toJpaEntity(course));
4 }
```



### Repositorio de escritura del ORM → Base de datos

```
1 public interface CourseEntityJpaRepository extends JpaRepository<CourseEntity, Integer> {  
2  
3 }
```

**NOTA:** El flujo de los eventos está explicado en la sección [5.2.10. Patrón saga](#).

### **5.2.17.5. Eliminar un curso**

Con **DELETE /courses/{id}**, el controlador llama a **courseServicePort.delete(id)**. El servicio elimina la entidad en PostgreSQL (**CourseCommandRepositoryPort.delete(id)**), publica **CourseDeletedEvent** para borrar la réplica en Redis y retorna un **ResponseEntity.ok()**.

### Controlador adaptador de entrada → Servicio

```
1 @DeleteMapping("/{id}")  
2 @PreAuthorize("hasRole(@role.INSTRUCTOR)")  
3 public ResponseEntity<Void> delete(@PathVariable Integer id) {  
4     courseServicePort.delete(id);  
5     return ResponseEntity.ok().build();  
6 }
```

### Servicio → Repositorio puerto de lectura

```
1 @Override  
2 @Transactional  
3 public void delete(Integer id) {  
4     Course oldCourse = getById(id);  
5  
6     courseCommandRepositoryPort.delete(id);  
7     eventPublisher.publishCourseDeleted(oldCourse);  
8 }
```



Repositorio adaptador de salida de escritura → Repositorio del ORM

```

1  @Override
2  public void delete(Integer id) {
3      courseEntityJpaRepository.deleteById(id);
4  }

```

Repositorio de escritura del ORM → Base de datos

```

1  public interface CourseEntityJpaRepository extends JpaRepository<CourseEntity, Integer> {
2
3 }

```

**NOTA:** El flujo de los eventos está explicado en la sección [5.2.10. Patrón saga](#).

**5.2.17.6. Inscribir un usuario en un curso**

La llamada **POST /courses/{courseId}/enrollments**, segura para estudiantes, extrae courseId, obtiene userId de **CurrentUserHelper** y ejecuta **courseServicePort.enroll(courseId, userId)**. El caso de uso valida reglas, persiste la inscripción en PostgreSQL (vía **EnrollmentCommandRepositoryPort.save(...)**), publica un evento de inscripción (si estuviera definido) y responde con **ResponseEntity.ok()**.

Controlador adaptador de entrada → Servicio

```

1  @PostMapping("/{courseId}/enrollments")
2  @PreAuthorize("hasRole(@role.STUDENT)")
3  public ResponseEntity<Void> enroll(@PathVariable Integer courseId) {
4      UUID userId = currentUserHelper.getId();
5      courseServicePort.enroll(courseId, userId);
6      return ResponseEntity.ok().build();
7  }

```



Servicio → Repositorio puerto de lectura

```

1  @Override
2  @Transactional
3  public void enroll(Integer courseId, UUID userId) throws UserAlreadyEnrolledInCourse {
4
5      Course course = getById(courseId);
6      Course oldCourse = cloner.copy(course, Course.class);
7
8      if (course.getEnrollments().stream()
9          .anyMatch(enrollment -> enrollment.getUser().getId().equals(userId))) {
10          throw new UserAlreadyEnrolledInCourse("User already enrolled in course with id: " +
11              courseId);
12      }
13
14      User user = userServicePort.getById(userId);
15
16      Enrollment enrollment = new Enrollment();
17      enrollment.enrollUser(user);
18      course.addEnrollments(List.of(enrollment));
19
20      courseCommandRepositoryPort.save(course);
21      eventPublisher.publishCourseUpdated(oldCourse, course);
22  }

```

Repositorio adaptador de salida de escritura → Repositorio del ORM

```

1  @Override
2  public void save(Course course) {
3      courseEntityJpaRepository.save(courseJpaMapper.toJpaEntity(course));
4  }

```

Repositorio de escritura del ORM → Base de datos

```

1  public interface CourseEntityJpaRepository extends JpaRepository<CourseEntity, Integer> {
2
3  }

```

**NOTA:** El flujo de los eventos está explicado en la sección [5.2.10. Patrón saga](#).

### 5.2.17.7. Completar una lección

La ruta POST /courses/{courseId}/sections/{sectionId}/lessons/{lessonId}/complete extrae los tres identificadores, obtiene el userId y llama a **courseServicePort.completeLesson(courseId, sectionId, lessonId, userId)**. El servicio registra el progreso en PostgreSQL y retorna **ResponseEntity.ok()**.

#### Controlador adaptador de entrada → Servicio

```
1 @PostMapping("/{courseId}/sections/{sectionId}/lessons/{lessonId}/complete")
2 @PreAuthorize("hasRole(@role.STUDENT)")
3 public ResponseEntity<Void> completeLesson(@PathVariable Integer courseId, @PathVariable Integer
4     sectionId, @PathVariable Integer lessonId) {
5     UUID userId = currentUserHelper.getId();
6     courseServicePort.completeLesson(courseId, sectionId, lessonId, userId);
7     return ResponseEntity.ok().build();
}
```

#### Servicio → Repositorio puerto de lectura

```
1 @Override
2 @Transactional
3 public void completeLesson(Integer courseId, Integer sectionId, Integer lessonId, UUID userId)
{
4
5     Course course = getById(courseId);
6     Course oldCourse = cloner.copy(course, Course.class);
7
8     User user = userServicePort.getById(userId);
9     Lesson lesson = lessonServicePort.getById(lessonId);
10    Section section = sectionServicePort.getById(sectionId);
11
12    if (!course.getSections().contains(section)) {
13        throw new ResourceNotFoundException("Section not found in course with id: " + courseId);
14    }
15
16    if (!section.getLessons().contains(lesson)) {
17        throw new ResourceNotFoundException("Lesson not found in section with id: " + sectionId);
18    }
19
20    Optional<Progress> maybeProgress = lesson.getProgress()
21        .stream()
22        .filter(progress -> progress.getUser().getId().equals(userId))
23        .findFirst();
24
25    Progress progress = maybeProgress.orElseGet(() -> {
26        Progress newProgress = new Progress();
27
28        newProgress.complete(user);
29
30        lesson.addProgresses(List.of(newProgress));
31        return newProgress;
32    });
33
34    lesson.addProgresses(List.of(progress));
35
36    courseCommandRepositoryPort.save(course);
37    eventPublisher.publishCourseUpdated(oldCourse, course);
38 }
```



### Repositorio adaptador de salida de escritura → Repositorio del ORM



```
1 @Override
2     public void save(Course course) {
3         courseEntityJpaRepository.save(courseJpaMapper.toJpaEntity(course));
4     }
```

### Repositorio de escritura del ORM → Base de datos



```
1 public interface CourseEntityJpaRepository extends JpaRepository<CourseEntity, Integer> {
2
3 }
```

**NOTA:** El flujo de los eventos está explicado en la sección [5.2.10. Patrón saga](#).

#### **5.2.17.8. ¿Qué sucede con el evento cuando un usuario se elimina en la app de auth?**

En el microservicio Go, al eliminar un usuario se dispara internamente un evento UserDeletedEvent en RabbitMQ mediante un EventPublisher. Dicho evento es consumido por un RabbitListener que, tras deserializarlo, invoca el método correspondiente en el servicio de dominio para borrar cualquier dato asociado en Redis o en otras colas, y finalmente envía un ACK al broker. De esta forma, la eliminación en la base principal (PostgreSQL) se propaga de forma asíncrona y consistente al sistema de cache o consulta rápido (Redis), manteniendo la coherencia eventual entre ambos almacenes.

Puede consultarse con más detalle en el apartado visto con anterioridad: [5.2.13. Consumidor de Eventos para RabbitMQ](#).



## 6. MANUAL DE USUARIO

En este capítulo se describen los pasos y herramientas necesarios para poner en marcha el entorno completo de GoLearnix, así como las indicaciones para verificar su correcto funcionamiento mediante Postman. Se detallan los requisitos previos, el método de ejecución de cada microservicio y, finalmente, una serie de pruebas sobre los distintos endpoints acompañadas de capturas de pantalla de las respuestas obtenidas.

### 6.1. Requisitos

Para poder ejecutar la plataforma GoLearnix, se requiere lo siguiente:

1. **Docker y Docker Compose**: para levantar rápidamente la infraestructura (PostgreSQL, Redis y RabbitMQ).
2. **Go 1.24.1**: compilador y entorno de ejecución para el microservicio de autenticación y gestión de usuarios. Se puede usar GoLand para ejecutar la aplicación.
3. **Java 21 y Maven 3.8**: JDK y gestor de dependencias/build para la aplicación de gestión de cursos. Se puede usar Inetllij para ejecutar la aplicación.
4. **Postman 11.46.2**: cliente HTTP para la ejecución de pruebas sobre la API REST.
5. **Git**: clonar el repositorio con el código fuente.

*Antes de iniciar el entorno, no debe haber puertos en uso en los rangos: 5432–5433 (PostgreSQL), 6379 (Redis), 5672–15672 (RabbitMQ), 2003 (Go) y 8080 (Java).*

### 6.2. Postman

Postman es una herramienta gráfica para desarrollar, probar y documentar APIs REST. Permite enviar peticiones HTTP (GET, POST, PUT, DELETE, etc.), parametrizar cabeceras y cuerpos JSON, guardar colecciones de peticiones y visualizar respuestas de forma ordenada. Su simplicidad y capacidad de scripting la convierten en el cliente ideal para validar rápidamente el comportamiento de cada endpoint.



## 6.3. Pruebas

A continuación, se describe el procedimiento de prueba para cada uno de los principales endpoints de la aplicación de cursos. Para facilitar este proceso, en el repositorio del proyecto se incluye una colección de Postman preconfigurada, que puede importarse directamente en la herramienta. Esta colección contiene todas las peticiones organizadas por funcionalidad (cursos y auth).

Una vez se ha iniciado Docker:

The screenshot shows the Docker Desktop interface. On the left, a sidebar has 'Containers' selected. The main area displays a table of running containers with columns: Name, Container ID, Image, Port(s), CPU (%), Last started, and Actions. The table lists five containers: golearnix, course-redis-db, rabbit, auth-psql-db, and course-psql-db. Each container's status and resource usage are shown. At the bottom, there are navigation links for 'Engine running', 'RAM 1.79 GB CPU 17.28% Disk: 15.23 GB used (limit 1006.85 GB)', and 'Terminal'.

Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
golearnix	-	-	-	2.88%	12 minutes ago	[Actions]
course-redis-db	6ae9ec194505	redis/redis-stack:7.4.0	6379:6379	2.29%	12 minutes ago	[Actions]
rabbit	019103ee4ae6	rabbitmq:3-management	15672:15672	0.58%	12 minutes ago	[Actions]
auth-psql-db	1125d8c06d05	postgres:17.4	5432:5432	0%	12 minutes ago	[Actions]
course-psql-db	caa05c3e9e3b	postgres:17.4	5433:5432	0.01%	12 minutes ago	[Actions]

Junto con la aplicación de Go:

The screenshot shows the GoLand IDE interface. The left panel shows a project structure for 'golearnix-auth' with files like 'datasource', 'docs', 'domain', 'events', 'presentation', 'env', 'go.mod', and 'go.sum'. The right panel is a terminal window displaying the output of a 'go build' command for 'golearnix-auth'. It shows the application is running on port 2003 with 13 handlers and 1 process. The terminal also shows a command to setup calls.

```
Fiber v2.52.6
http://127.0.0.1:2003
(bound on host 0.0.0.0 and port 2003)

Handlers ..... 13 Processes ..... 1
Prefork ..... Disabled PID ..... 9544
```

## Y la aplicación de Java:

```

2025-05-23T12:30:32.079+02:00 INFO 9128 --- [golearnix] c.r.o.spring.indexing.RedisSearchIndexer : Using entity prefix 'enrollment:' as keyspace for type : class com.golearnix.redis.entities.EnrollmentReadModelIdx
2025-05-23T12:30:32.087+02:00 INFO 9128 --- [golearnix] c.r.o.spring.indexing.RedisSearchIndexer : Created index com.golearnix.redis.entities.EnrollmentReadModelIdx
2025-05-23T12:30:32.209+02:00 INFO 9128 --- [golearnix] c.r.o.spring.indexing.RedisSearchIndexer : Found 0 @RedisHash annotated Beans...
2025-05-23T12:30:32.533+02:00 INFO 9128 --- [golearnix] c.r.o.spring.indexing.RedisSearchIndexer : Started Application in 53.485 seconds (process running for 57.994)
2025-05-23T12:30:33.302+02:00 INFO 9128 --- [golearnix] o.a.c.c.c.Tomcat : Initializing Spring DispatcherServlet 'dispatcherServlet'
2025-05-23T12:30:33.302+02:00 INFO 9128 --- [golearnix] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2025-05-23T12:30:33.308+02:00 INFO 9128 --- [golearnix] o.s.web.servlet.DispatcherServlet : Completed initialization in 6 ms

```

Se pueden realizar las comprobaciones. Además, en el acceso web de **RabbitMQ** se puede comprobar:

### Las conexiones de Go y Java:

Name	User name	State	SSL / TLS	Protocol	Channels	From client	To client
172.19.0.1:36202	golearnix	running	o	AMQP 0-9-1	1	2 B/s	2 B/s
172.19.0.1:50696	golearnix	running	o	AMQP 0-9-1	2	0 B/s	0 B/s

### Las creaciones de los Exchange:

Virtual host	Name	Type	Features	Message rate in	Message rate out
/	(AMQP default)	direct	D		
/	amq.direct	direct	D		
/	amq.fanout	fanout	D		
/	amq.headers	headers	D		
/	amq.match	headers	D		
/	amq.rabbitmq.trace	topic	D I		
/	amq.topic	topic	D		
/	golearnix.events	topic	D AD		
/	golearnix.events.dlx	topic	D AD		



Las colas (Queue):
**1- Registrarse:**

```

1 {
2   "name": "Juan Pérez",
3   "email": "juan@example.com",
4   "password": "supersegura123",
5   "role": "student"
6 }
    
```

```

1 {
2   "success": true,
3   "message": "Usuario registrado exitosamente"
4 }
    
```

**2- Iniciar sesión:**

```

1 {
2   "email": "juan@example.com",
3   "password": "supersegura123"
4 }
    
```

```

1 {
2   "success": true,
3   "message": "Inicio de sesión exitoso",
4   "data": {
5     "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
       eyJzdWIiOiI4N2M0NTjy11NjI8LjQ4NDMtQ0QwMy00MmUyZnI0YjFlMGU1LC3yb2xIjoic3R1ZGVudCisImV4cCI6MTe8ODA4NDc3NSwiWF0IjoxNzQ3OTk4Mzc1LCJqdGkiOiIzNjgyZTBnMy84ZmM2LTQzWnItYiYzI84MmMyZTkwZTJ1M2Q1fQ.wzov1uqVLDFfcSzLNDoDn4_685IR81XwJ856BGQHesp0"
6   }
7 }
    
```



CIPFP Mislata

Centre Integrat Públic  
Formació Professional Superior

- 3- Introducir el token devuelto por iniciar sesión:** Para ello hacer click en *GoLearnix* → *Autorization* → *Bearer Token*. Las demás acciones heredan de la autorización del padre para no tener que repetir el proceso en cada acción.

The screenshot shows the Postman interface with the 'GoLearnix' collection selected. In the 'Authorization' tab, 'Bearer Token' is chosen. The 'Token' field contains '\*\*\*\*\*'. The left sidebar shows other collections like 'Biblioteca' and 'Elevate'.

#### 4- Validar la sesión / cuenta:

The screenshot shows a specific API call within the 'GoLearnix' collection. It's a 'GET' request to 'localhost:2003/api/v1/auth/validate'. The 'Body' tab displays a JSON response:

```

1 {
2     "success": true,
3     "message": "Token válido",
4     "data": {
5         "email": "juan@example.com",
6         "expires_in": "23h:57m"
7     }
8 }

```



## 5- Ver los detalles del usuario:

The screenshot shows the Postman interface with the following details:

- Collection:** Personal Workspace
- API:** GET Auth - Información del user
- URL:** localhost:2003/api/v1/user/me
- Method:** GET
- Body:** JSON (Preview tab)
- Response:**

```

1 {
2     "success": true,
3     "message": "Información del usuario",
4     "data": [
5         "user": {
6             "id": "87c493cb-e654-4843-8d83-42e2fb4b1e0e",
7             "name": "Juan Pérez",
8             "email": "juan@example.com",
9             "role": "student",
10            "created_at": "2025-05-23T13:04:49.942469+02:00",
11            "updated_at": "2025-05-23T13:04:49.942469+02:00"
12        }
13    ]
14 }

```

## 6- Cerrar la sesión:

The screenshot shows the Postman interface with the following details:

- Collection:** Personal Workspace
- API:** POST Auth - Cerrar Sesión
- URL:** localhost:2003/api/v1/auth/logout
- Method:** POST
- Body:** JSON (Preview tab)
- Response:**

```

1 {
2     "success": true,
3     "message": "Sesión cerrada exitosamente"
4 }

```

## 7- Ver todos los cursos:

The screenshot shows the Postman interface with the following details:

- Collection:** Personal Workspace
- API:** GET Obtener todos los cursos
- URL:** localhost:8080/api/v1/golearnix/courses
- Method:** GET
- Body:** JSON (Preview tab)
- Response:**

```

1 [
2     {
3         "description": "Learn Go from beginner to expert.",
4         "title": "Go Masterclass"
5     },
6     {
7         "description": "Fundamentals of UX and user experience.",
8         "title": "Introduction to UX Design"
9     },
10    {
11        "description": "Learn the basics of digital marketing.",
12        "title": "Digital Marketing 101"
13    },
14    {
15        "description": "Build and manage a startup business.",
16        "title": "Startup Fundamentals"
17    }
18 ]

```



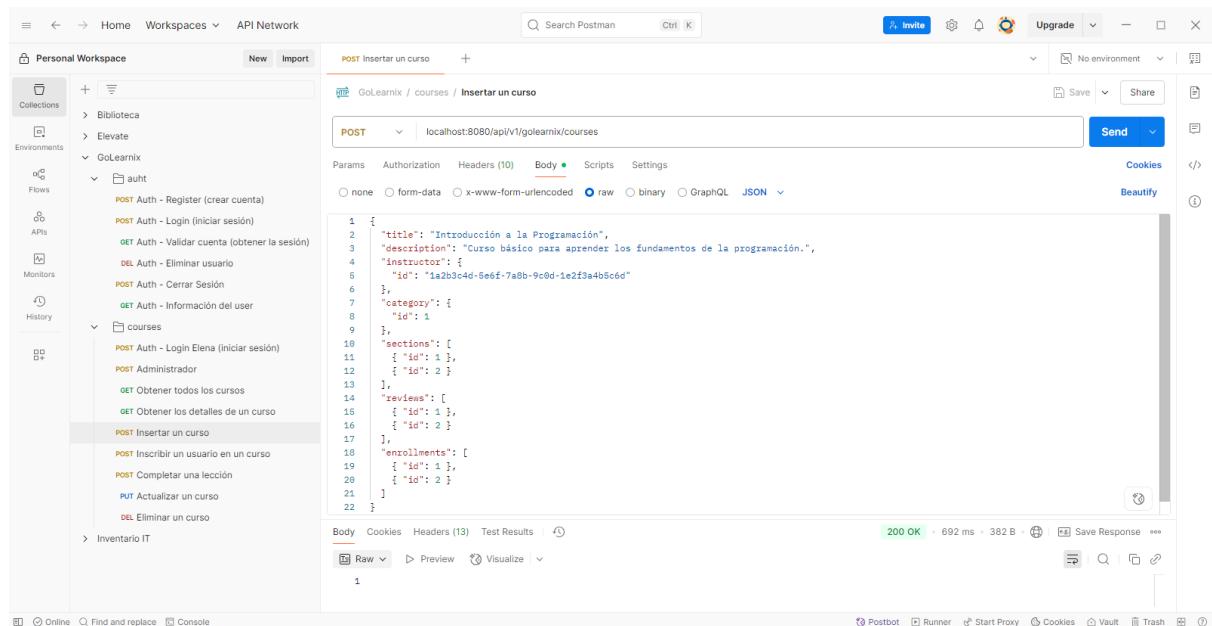
## 8- Ver los detalles de un curso:

The screenshot shows the Postman interface with a collection named "Golearnix". A specific API endpoint, "Obtener todos los cursos", is selected. The request method is GET, and the URL is localhost:8080/api/v1/golearnix/courses. The response status is 200 OK, and the response body is a JSON array of course objects:

```

1 [
2   {
3     "id": 1,
4     "title": "Go Masterclass",
5     "description": "Learn Go from beginner to expert.",
6     "instructor": {
7       "id": "e709af2f-3c4d-4a5f-9a2b-7c8d9e0f1a2b"
8     },
9     "category": {
10       "id": 1,
11       "name": "Programación",
12       "description": "Software development courses."
13     },
14     "sections": [
15       {
16         "id": 2,
17         "title": "Go Best Practices",
18         "order": 2,
19         "lessons": [
20           {
21             "id": 3,
22             "title": "Error Handling",
23             "order": 1,
24             "content": "Handling errors and panic/recover.",
25             "progress": {}
26           }
27         ],
28         {
29           "id": 1,
30           "title": "Go Fundamentals",
31           "order": 1,
32           "lessons": [
33             {
34               "id": 2,
35               "title": "Types and Variables",
36               "order": 2,
37               "content": "Working with types and variables in Go.",
38               "progress": [
39                 {
40                   "id": 2,
41                   "user": {
42                     "id": "0f1e2d3c-4b5a-6978-8c7d-6e5f4a3b2c1d"
43                   },
44                   "completed": true
45                 }
46               ],
47               {
48                 "id": 1,
49                 "title": "Basic Syntax",
50                 "order": 1,
51                 "content": "Introduction to Go syntax.",
52                 "progress": [
53                   {
54                     "id": 3,
55                     "user": {
56                       "id": "1a2b3c4d-5e6f-7a8b-9c0d-1e2f3a4b5c6d"
57                     },
58                     "completed": false
59                   },
60                   {
61                     "id": 1,
62                     "user": {
63                       "id": "0f1e2d3c-4b5a-6978-8c7d-6e5f4a3b2c1d"
64                     },
65                     "completed": true
66                   }
67                 ]
68               }
69             ],
70           }
71         ],
72         "reviews": [
73           {
74             "id": 1,
75             "user": {
76               "id": "0f1e2d3c-4b5a-6978-8c7d-6e5f4a3b2c1d"
77             },
78             "rating": 5,
79             "comment": "Excellent Go course!"
80           },
81           {
82             "id": 2,
83             "user": {
84               "id": "1a2b3c4d-5e6f-7a8b-9c0d-1e2f3a4b5c6d"
85             },
86             "rating": 4,
87             "comment": "Very good, helped me a lot."
88           }
89         ],
90         "enrollments": [
91           {
92             "id": 1,
93             "user": {
94               "id": "0f1e2d3c-4b5a-6978-8c7d-6e5f4a3b2c1d"
95             }
96           },
97           {
98             "id": 2,
99             "user": {
100               "id": "1a2b3c4d-5e6f-7a8b-9c0d-1e2f3a4b5c6d"
101             }
102           }
103         ]
104       ]
105     ]
106   ]
107 ]
108 ]
109 ]
110 ]
111 ]
112 ]
113 ]
114 ]
115 ]
116 ]
117 ]
118 ]
119 ]
120 ]
121 ]
122 ]
123 ]
124 ]
125 ]
126 ]
127 ]
128 ]
129 ]
130 ]
131 ]
132 ]
133 ]
134 ]
135 ]
136 ]
137 ]
138 ]
139 ]
140 ]
141 ]
142 ]
143 ]
144 ]
145 ]
146 ]
147 ]
148 ]
149 ]
150 ]
151 ]
152 ]
153 ]
154 ]
155 ]
156 ]
157 ]
158 ]
159 ]
160 ]
161 ]
162 ]
163 ]
164 ]
165 ]
166 ]
167 ]
168 ]
169 ]
170 ]
171 ]
172 ]
173 ]
174 ]
175 ]
176 ]
177 ]
178 ]
179 ]
180 ]
181 ]
182 ]
183 ]
184 ]
185 ]
186 ]
187 ]
188 ]
189 ]
190 ]
191 ]
192 ]
193 ]
194 ]
195 ]
196 ]
197 ]
198 ]
199 ]
199 ]
200 ]
201 ]
202 ]
203 ]
204 ]
205 ]
206 ]
207 ]
208 ]
209 ]
209 ]
210 ]
211 ]
212 ]
213 ]
214 ]
215 ]
216 ]
217 ]
218 ]
219 ]
219 ]
220 ]
221 ]
222 ]
223 ]
224 ]
225 ]
226 ]
227 ]
228 ]
229 ]
229 ]
230 ]
231 ]
232 ]
233 ]
234 ]
235 ]
236 ]
237 ]
238 ]
239 ]
239 ]
240 ]
241 ]
242 ]
243 ]
244 ]
245 ]
246 ]
247 ]
247 ]
248 ]
249 ]
249 ]
250 ]
251 ]
252 ]
253 ]
254 ]
255 ]
256 ]
256 ]
257 ]
258 ]
259 ]
259 ]
260 ]
261 ]
262 ]
263 ]
264 ]
265 ]
266 ]
267 ]
267 ]
268 ]
269 ]
269 ]
270 ]
271 ]
272 ]
273 ]
274 ]
275 ]
276 ]
277 ]
278 ]
279 ]
279 ]
280 ]
281 ]
282 ]
283 ]
284 ]
285 ]
286 ]
287 ]
288 ]
289 ]
289 ]
290 ]
291 ]
292 ]
293 ]
294 ]
295 ]
296 ]
297 ]
298 ]
299 ]
299 ]
300 ]
301 ]
302 ]
303 ]
304 ]
305 ]
306 ]
307 ]
308 ]
309 ]
309 ]
310 ]
311 ]
312 ]
313 ]
314 ]
315 ]
316 ]
317 ]
318 ]
319 ]
319 ]
320 ]
321 ]
322 ]
323 ]
324 ]
325 ]
326 ]
327 ]
328 ]
329 ]
329 ]
330 ]
331 ]
332 ]
333 ]
334 ]
335 ]
336 ]
337 ]
338 ]
339 ]
339 ]
340 ]
341 ]
342 ]
343 ]
344 ]
345 ]
346 ]
347 ]
348 ]
349 ]
349 ]
350 ]
351 ]
352 ]
353 ]
354 ]
355 ]
356 ]
357 ]
358 ]
359 ]
359 ]
360 ]
361 ]
362 ]
363 ]
364 ]
365 ]
366 ]
367 ]
368 ]
369 ]
369 ]
370 ]
371 ]
372 ]
373 ]
374 ]
375 ]
376 ]
377 ]
378 ]
379 ]
379 ]
380 ]
381 ]
382 ]
383 ]
384 ]
385 ]
386 ]
387 ]
388 ]
389 ]
389 ]
390 ]
391 ]
392 ]
393 ]
394 ]
395 ]
396 ]
397 ]
398 ]
399 ]
399 ]
400 ]
401 ]
402 ]
403 ]
404 ]
405 ]
406 ]
407 ]
408 ]
409 ]
409 ]
410 ]
411 ]
412 ]
413 ]
414 ]
415 ]
416 ]
417 ]
418 ]
419 ]
419 ]
420 ]
421 ]
422 ]
423 ]
424 ]
425 ]
426 ]
427 ]
428 ]
429 ]
429 ]
430 ]
431 ]
432 ]
433 ]
434 ]
435 ]
436 ]
437 ]
438 ]
439 ]
439 ]
440 ]
441 ]
442 ]
443 ]
444 ]
445 ]
446 ]
447 ]
448 ]
449 ]
449 ]
450 ]
451 ]
452 ]
453 ]
454 ]
455 ]
456 ]
457 ]
458 ]
459 ]
459 ]
460 ]
461 ]
462 ]
463 ]
464 ]
465 ]
466 ]
467 ]
468 ]
469 ]
469 ]
470 ]
471 ]
472 ]
473 ]
474 ]
475 ]
476 ]
477 ]
478 ]
478 ]
479 ]
480 ]
481 ]
482 ]
483 ]
484 ]
485 ]
486 ]
487 ]
488 ]
488 ]
489 ]
489 ]
490 ]
491 ]
492 ]
493 ]
494 ]
495 ]
496 ]
497 ]
498 ]
499 ]
499 ]
500 ]
501 ]
502 ]
503 ]
504 ]
505 ]
506 ]
507 ]
508 ]
509 ]
509 ]
510 ]
511 ]
512 ]
513 ]
514 ]
515 ]
516 ]
517 ]
518 ]
519 ]
519 ]
520 ]
521 ]
522 ]
523 ]
524 ]
525 ]
526 ]
527 ]
528 ]
529 ]
529 ]
530 ]
531 ]
532 ]
533 ]
534 ]
535 ]
536 ]
537 ]
538 ]
539 ]
539 ]
540 ]
541 ]
542 ]
543 ]
544 ]
545 ]
546 ]
547 ]
548 ]
549 ]
549 ]
550 ]
551 ]
552 ]
553 ]
554 ]
555 ]
556 ]
557 ]
558 ]
559 ]
559 ]
560 ]
561 ]
562 ]
563 ]
564 ]
565 ]
566 ]
567 ]
568 ]
569 ]
569 ]
570 ]
571 ]
572 ]
573 ]
574 ]
575 ]
576 ]
577 ]
578 ]
579 ]
579 ]
580 ]
581 ]
582 ]
583 ]
584 ]
585 ]
586 ]
587 ]
588 ]
589 ]
589 ]
590 ]
591 ]
592 ]
593 ]
594 ]
595 ]
596 ]
597 ]
598 ]
599 ]
599 ]
600 ]
601 ]
602 ]
603 ]
604 ]
605 ]
606 ]
607 ]
608 ]
609 ]
609 ]
610 ]
611 ]
612 ]
613 ]
614 ]
615 ]
616 ]
617 ]
618 ]
619 ]
619 ]
620 ]
621 ]
622 ]
623 ]
624 ]
625 ]
626 ]
627 ]
628 ]
629 ]
629 ]
630 ]
631 ]
632 ]
633 ]
634 ]
635 ]
636 ]
637 ]
638 ]
639 ]
639 ]
640 ]
641 ]
642 ]
643 ]
644 ]
645 ]
646 ]
647 ]
648 ]
649 ]
649 ]
650 ]
651 ]
652 ]
653 ]
654 ]
655 ]
656 ]
657 ]
658 ]
659 ]
659 ]
660 ]
661 ]
662 ]
663 ]
664 ]
665 ]
666 ]
667 ]
668 ]
669 ]
669 ]
670 ]
671 ]
672 ]
673 ]
674 ]
675 ]
676 ]
677 ]
678 ]
679 ]
679 ]
680 ]
681 ]
682 ]
683 ]
684 ]
685 ]
686 ]
687 ]
688 ]
689 ]
689 ]
690 ]
691 ]
692 ]
693 ]
694 ]
695 ]
696 ]
697 ]
698 ]
699 ]
699 ]
700 ]
701 ]
702 ]
703 ]
704 ]
705 ]
706 ]
707 ]
708 ]
709 ]
709 ]
710 ]
711 ]
712 ]
713 ]
714 ]
715 ]
716 ]
717 ]
718 ]
719 ]
719 ]
720 ]
721 ]
722 ]
723 ]
724 ]
725 ]
726 ]
727 ]
728 ]
729 ]
729 ]
730 ]
731 ]
732 ]
733 ]
734 ]
735 ]
736 ]
737 ]
738 ]
739 ]
739 ]
740 ]
741 ]
742 ]
743 ]
744 ]
745 ]
746 ]
747 ]
748 ]
749 ]
749 ]
750 ]
751 ]
752 ]
753 ]
754 ]
755 ]
756 ]
757 ]
758 ]
759 ]
759 ]
760 ]
761 ]
762 ]
763 ]
764 ]
765 ]
766 ]
767 ]
768 ]
769 ]
769 ]
770 ]
771 ]
772 ]
773 ]
774 ]
775 ]
776 ]
777 ]
778 ]
778 ]
779 ]
779 ]
780 ]
781 ]
782 ]
783 ]
784 ]
785 ]
785 ]
786 ]
786 ]
787 ]
787 ]
788 ]
788 ]
789 ]
789 ]
790 ]
791 ]
792 ]
793 ]
794 ]
795 ]
796 ]
797 ]
798 ]
799 ]
799 ]
800 ]
801 ]
802 ]
803 ]
804 ]
805 ]
806 ]
807 ]
808 ]
809 ]
809 ]
810 ]
811 ]
812 ]
813 ]
814 ]
815 ]
816 ]
817 ]
818 ]
819 ]
819 ]
820 ]
821 ]
822 ]
823 ]
824 ]
825 ]
826 ]
827 ]
828 ]
829 ]
829 ]
830 ]
831 ]
832 ]
833 ]
834 ]
835 ]
836 ]
837 ]
838 ]
839 ]
839 ]
840 ]
841 ]
842 ]
843 ]
844 ]
845 ]
846 ]
847 ]
848 ]
849 ]
849 ]
850 ]
851 ]
852 ]
853 ]
854 ]
855 ]
856 ]
857 ]
858 ]
859 ]
859 ]
860 ]
861 ]
862 ]
863 ]
864 ]
865 ]
866 ]
867 ]
868 ]
869 ]
869 ]
870 ]
871 ]
872 ]
873 ]
874 ]
875 ]
876 ]
877 ]
878 ]
878 ]
879 ]
879 ]
880 ]
881 ]
882 ]
883 ]
884 ]
885 ]
886 ]
887 ]
888 ]
888 ]
889 ]
889 ]
890 ]
891 ]
892 ]
893 ]
894 ]
895 ]
896 ]
897 ]
898 ]
899 ]
899 ]
900 ]
901 ]
902 ]
903 ]
904 ]
905 ]
906 ]
907 ]
908 ]
909 ]
909 ]
910 ]
911 ]
912 ]
913 ]
914 ]
915 ]
916 ]
917 ]
918 ]
919 ]
919 ]
920 ]
921 ]
922 ]
923 ]
924 ]
925 ]
926 ]
927 ]
928 ]
929 ]
929 ]
930 ]
931 ]
932 ]
933 ]
934 ]
935 ]
936 ]
937 ]
938 ]
939 ]
939 ]
940 ]
941 ]
942 ]
943 ]
944 ]
945 ]
946 ]
947 ]
948 ]
949 ]
949 ]
950 ]
951 ]
952 ]
953 ]
954 ]
955 ]
956 ]
957 ]
958 ]
959 ]
959 ]
960 ]
961 ]
962 ]
963 ]
964 ]
965 ]
966 ]
967 ]
968 ]
969 ]
969 ]
970 ]
971 ]
972 ]
973 ]
974 ]
975 ]
976 ]
977 ]
978 ]
978 ]
979 ]
979 ]
980 ]
981 ]
982 ]
983 ]
984 ]
985 ]
986 ]
987 ]
988 ]
988 ]
989 ]
989 ]
990 ]
991 ]
992 ]
993 ]
994 ]
995 ]
996 ]
997 ]
998 ]
999 ]
999 ]
1000 ]
1001 ]
1002 ]
1003 ]
1004 ]
1005 ]
1006 ]
1007 ]
1008 ]
1009 ]
1009 ]
1010 ]
1011 ]
1012 ]
1013 ]
1014 ]
1015 ]
1016 ]
1017 ]
1018 ]
1019 ]
1019 ]
1020 ]
1021 ]
1022 ]
1023 ]
1024 ]
1025 ]
1026 ]
1027 ]
1028 ]
1029 ]
1029 ]
1030 ]
1031 ]
1032 ]
1033 ]
1034 ]
1035 ]
1036 ]
1037 ]
1038 ]
1039 ]
1039 ]
1040 ]
1041 ]
1042 ]
1043 ]
1044 ]
1045 ]
1046 ]
1047 ]
1048 ]
1049 ]
1049 ]
1050 ]
1051 ]
1052 ]
1053 ]
1054 ]
1055 ]
1056 ]
1057 ]
1058 ]
1059 ]
1059 ]
1060 ]
1061 ]
1062 ]
1063 ]
1064 ]
1065 ]
1066 ]
1067 ]
1068 ]
1069 ]
1069 ]
1070 ]
1071 ]
1072 ]
1073 ]
1074 ]
1075 ]
1076 ]
1077 ]
1078 ]
1078 ]
1079 ]
1079 ]
1080 ]
1081 ]
1082 ]
1083 ]
1084 ]
1085 ]
1086 ]
1087 ]
1088 ]
1088 ]
1089 ]
1089 ]
1090 ]
1091 ]
1092 ]
1093 ]
1094 ]
1095 ]
1095 ]
1096 ]
1096 ]
1097 ]
1098 ]
1099 ]
1099 ]
1100 ]
1101 ]
1102 ]
1103 ]
1104 ]
1105 ]
1106 ]
1107 ]
1108 ]
1109 ]
1109 ]
1110 ]
1111 ]
1112 ]
1113 ]
1114 ]
1115 ]
1116 ]
1117 ]
1118 ]
1119 ]
1119 ]
1120 ]
1121 ]
1122 ]
1123 ]
1124 ]
1125 ]
1126 ]
1127 ]
1128 ]
1129 ]
1129 ]
1130 ]
1131 ]
1132 ]
1133 ]
1134 ]
1135 ]
1136 ]
1137 ]
1138 ]
1138 ]
1139 ]
1139 ]
1140 ]
1141 ]
1142 ]
1143 ]
1144 ]
1145 ]
1146 ]
1147 ]
1148 ]
1148 ]
1149 ]
1149 ]
1150 ]
1151 ]
1152 ]
1153 ]
1154 ]
1155 ]
1156 ]
1157 ]
1158 ]
1159 ]
1159 ]
1160 ]
1161 ]
1162 ]
1163 ]
1164 ]
1165 ]
1166 ]
1167 ]
1168 ]
1169 ]
1169 ]
1170 ]
1171 ]
1172 ]
1173 ]
1174 ]
1175 ]
1176 ]
1177 ]
1178 ]
1178 ]
1179 ]
1179 ]
1180 ]
1181 ]
1182 ]
1183 ]
1184 ]
1185 ]
1186 ]
1187 ]
1188 ]
1188 ]
1189 ]
1189 ]
1190 ]
1191 ]
1192 ]
1193 ]
1194 ]
1195 ]
1195 ]
1196 ]
1196 ]
1197 ]
1198 ]
1199 ]
1199 ]
1200 ]
1201 ]
1202 ]
1203 ]
1204 ]
1205 ]
1206 ]
1207 ]
1208 ]
1209 ]
1209 ]
1210 ]
1211 ]
1212 ]
1213 ]
1214 ]
1215 ]
1216 ]
1217 ]
1218 ]
1218 ]
1219 ]
1219 ]
1220 ]
1221 ]
1222 ]
1223 ]
1224 ]
1225 ]
1226 ]
1227 ]
1228 ]
1228 ]
1229 ]
1229 ]
1230 ]
1231 ]
1232 ]
1233 ]
1234 ]
1235 ]
1236 ]
1237 ]
1238 ]
1238 ]
1239 ]
1239 ]
1240 ]
1241 ]
1242 ]
1243 ]
1244 ]
1245 ]
1246 ]
1247 ]
1248 ]
1248 ]
1249 ]
1249 ]
1250 ]
1251 ]
1252 ]
1253 ]
1254 ]
1255 ]
1256 ]
1257 ]
1258 ]
1258 ]
1259 ]
1259 ]
1260 ]
1261 ]
1262 ]
1263 ]
1264 ]
1265 ]
1266 ]
1267 ]
1268 ]
1268 ]
1269 ]
1269 ]
1270 ]
1271 ]
1272 ]
1273 ]
1274 ]
1275 ]
1276 ]
1277 ]
1277 ]
1278 ]
1278 ]
1279 ]
1279 ]
1280 ]
1281 ]
1282 ]
1283 ]
1284 ]
1285 ]
1286 ]
1287 ]
1287 ]
1288 ]
1288 ]
1289 ]
1289 ]
1290 ]
1291 ]
1292 ]
1293 ]
1294 ]
1295 ]
1295 ]
1296 ]
1296 ]
1297 ]
1298 ]
1299 ]
1299 ]
1300 ]
1301 ]
1302 ]
1303 ]
1304 ]
1305 ]
1306 ]
1307 ]
1308 ]
1309 ]
1309 ]
1310 ]
1311 ]
1312 ]
1313 ]
1314 ]
1315 ]
1316 ]
1317 ]
1318 ]
1318 ]
1319 ]
1319 ]
1320 ]
1321 ]
1322 ]
1323 ]
1324 ]
1325 ]
1326 ]
1327 ]
1328 ]
1328 ]
1329 ]
1329 ]
1330 ]
1331 ]
1332 ]
1333 ]
1334 ]
1335 ]
1336 ]
1337 ]
1338 ]
1338 ]
1339 ]
1339 ]
1340 ]
1341 ]
1342 ]
1343 ]
1344 ]
1345 ]
1346 ]
1347 ]
1348 ]
1348 ]
1349 ]
1349 ]
1350 ]
1351 ]
1352 ]
1353 ]
1354 ]
1355 ]
1356 ]
1357 ]
1358 ]
1358 ]
1359 ]
1359 ]
1360 ]
1361 ]
1362 ]
1363 ]
1364 ]
1365 ]
1366 ]
1367 ]
1368 ]
1368 ]
1369 ]
1369 ]
1370 ]
1371 ]
1372 ]
1373 ]
1374 ]
1375 ]
1376 ]
1377 ]
1377 ]
1378 ]
1378 ]
1379 ]
1379 ]
1380 ]
1381 ]
1382 ]
1383 ]
1384 ]
1385 ]
1386 ]
1387 ]
1388 ]
1388 ]
1389 ]
1389 ]
1390 ]
1391 ]
1392 ]
1393 ]
1394 ]
1395 ]
1396 ]
1397 ]
1398 ]
1398 ]
1399 ]
1399 ]
1400 ]
1401 ]
1402 ]
1403 ]
1404 ]
1405 ]
1406 ]
1407 ]
1408 ]
1409 ]
1409 ]
1410 ]
1411 ]
1412 ]
1413 ]
1414 ]
1415 ]
1416 ]
1417 ]
1418 ]
1418 ]
1419 ]
1419 ]
1420 ]
1421 ]
1422 ]
1423 ]
1424 ]
1425 ]
1426 ]
1427 ]
1428 ]
1428 ]
1429 ]
1429 ]
1430 ]
1431 ]
1432 ]
1433 ]
1434 ]
1435 ]
1436 ]
1437 ]
1438 ]
1438 ]
1439 ]
1439 ]
1440 ]
1441 ]
1442 ]
1443 ]
1444 ]
1445 ]
1446 ]
1447 ]
1448 ]
1448 ]
1449 ]
1449 ]
1450 ]
1451 ]
1452 ]
1453 ]
1454 ]
1455 ]
1456 ]
1457 ]
1458 ]
1458 ]
1459 ]
1459 ]
1460 ]
1461 ]
1462 ]
1463 ]
1464 ]
1465 ]
1466 ]
1467 ]
1468 ]
1468 ]
1469 ]
1469 ]
1470 ]
1471 ]
1472 ]
1473 ]
1474 ]
1475 ]
1476 ]
1477 ]
1477 ]
1478 ]
1478 ]
1479 ]
1479 ]
1480 ]
1481 ]
1482 ]
1483 ]
1484 ]
1485 ]
1486 ]
1487 ]
1488 ]
1488 ]
1489 ]
1489 ]
1490 ]
1491 ]
1492 ]
1493 ]
1494 ]
1495 ]
1496 ]
1497 ]
1498 ]
1498 ]
1499 ]
1499 ]
1500 ]
1501 ]
1502 ]
1503 ]
1504 ]
1505 ]
1506 ]
1507 ]
1508 ]
1509 ]
1509 ]
1510 ]
1511 ]
1512 ]
1513 ]
1514 ]
1515 ]
1516 ]
1517 ]
1518 ]
1518 ]
1519 ]
1519 ]
1520 ]
1521 ]
1522 ]
1523 ]
1524 ]
1525 ]
1526 ]
1527 ]
1528 ]
1528 ]
1529 ]
1529 ]
1530 ]
1531 ]
1532 ]
1533 ]
1534 ]
1535 ]
1536 ]
1537 ]
1538 ]
1538 ]
1539 ]
1539 ]
1540 ]
1541 ]
1542 ]
1543 ]
1544 ]
1545 ]
1546 ]
1547 ]
1548 ]
1548 ]
1549 ]
1549 ]
1550 ]
1551 ]
1552 ]
1553 ]
1554 ]
1555 ]
1556 ]
1557 ]
1558 ]
1558 ]
1559 ]
1559 ]
1560 ]
1561 ]
1562 ]
1563 ]
1564 ]
1565 ]
1566 ]
1567 ]
1568 ]
1568 ]
1569 ]
1569 ]
1570 ]
1571 ]
1572 ]
1573 ]
1574 ]
1575 ]
1576 ]
1577 ]
1577 ]
1578 ]
1578 ]
1579 ]
1579 ]
1580 ]
1581 ]
1582 ]
1583 ]
1584 ]
1585 ]
1586 ]
1587 ]
1588 ]
1588 ]
1589 ]
1589 ]
1590 ]
1591 ]
1592 ]
1593 ]
1594 ]
1595 ]
1596 ]
1597 ]
1598 ]
1598 ]
1599 ]
1599 ]
1600 ]
1601 ]
1602 ]
1603 ]
1604 ]
1605 ]
1606 ]
1607 ]
1608 ]
1609 ]
1609 ]
1610 ]
1611 ]
1612 ]
1613 ]
1614 ]
1615 ]
1616 ]
1617 ]
1618 ]
1618 ]
1619 ]
1619 ]
1620 ]
1621 ]
1622 ]
1623 ]
1624 ]
1625 ]
1626 ]
1627 ]
1628 ]
1628 ]
1629 ]
1629 ]
1630 ]
1631 ]
1632 ]
1633 ]
1634 ]
1635 ]
1636 ]
1637 ]
1638 ]
1638 ]
1639 ]
1639 ]
1640 ]
1641 ]
1642 ]
1643 ]
1644 ]
1645 ]
1646 ]
1647 ]
1648 ]
1648 ]
1649 ]
1649 ]
1650 ]
1651 ]
1652 ]
1653 ]
1654 ]
1655 ]
1656 ]
1657 ]
1658 ]
1658 ]
1659 ]
1659 ]
1660 ]
1661 ]
1662 ]
1663 ]
1664 ]
1665 ]
1666 ]
1667 ]
1668 ]
1668 ]
1669 ]
1669 ]
1670 ]
1671 ]
1672 ]
1673 ]
1674 ]
1675 ]
1676 ]
1677 ]
1677 ]
1678 ]
1678 ]
1679 ]
1679 ]
1680 ]
1681 ]
1682 ]
1683 ]
1684 ]
1685 ]
1686 ]
1687 ]
1688 ]
1688 ]
1689 ]
1689 ]
1690 ]
1691 ]
1692 ]
1693 ]
1694 ]
1695 ]
1696 ]
1697 ]
1698 ]
1698 ]
1699 ]
1699 ]
1700 ]
1701 ]
1702 ]
1703 ]
1704 ]
1705 ]
1706 ]
1707 ]
1708 ]
1709 ]
1709 ]
1710 ]
1711 ]
1712 ]
1713 ]
1714 ]
1715 ]
1716 ]
1717 ]
1718 ]
1718 ]
1719 ]
1719 ]
1720 ]
1721 ]
1722 ]
1723 ]
1724 ]
1725 ]
1726 ]
1727 ]
1728 ]
1728 ]
1729 ]
1729 ]
1730 ]
1731 ]
1732 ]
1733 ]
1734 ]
1735 ]
1736 ]
1737 ]
1738 ]
1738 ]
1739 ]
1739 ]
1740 ]
1741 ]
1742 ]
1743 ]
1744 ]
1745 ]
1746 ]
1747 ]
1748 ]
1748 ]
1749 ]
1749 ]
1750 ]
1751 ]
1752 ]
1753 ]
1754 ]
1755 ]
1756 ]
1757 ]
1758 ]
1758 ]
1759 ]
1759 ]
1760 ]
1761 ]
1762 ]
1763 ]
1764 ]
1765 ]
1766 ]
1767 ]
1768 ]
1768 ]
1769 ]
1769 ]
1770 ]
1771 ]
1772 ]
1773 ]
1774 ]
1775 ]
1776 ]
1777 ]
1778 ]
1778 ]
1779 ]
1779 ]
1780 ]
1781 ]
1782 ]
1783 ]
1784 ]
1785 ]
1786 ]
1787 ]
1788 ]
1788 ]
1789 ]
1789 ]
1790 ]
1791 ]
1792 ]
1793 ]
1794 ]
1795 ]
1796 ]
1797 ]
1798 ]
1798 ]
1799 ]
1799 ]
1800 ]
1801 ]
1802 ]
1803 ]
1804 ]
1805 ]
1806 ]
1807 ]
1808 ]
1809 ]
1809 ]
1810 ]
1811 ]
1812 ]
1813 ]
1814 ]
1815 ]
1816 ]
1817 ]
1818 ]
1818 ]
1819 ]
1819 ]
1820 ]
1821 ]
1822 ]
1823 ]
1824 ]
1825 ]
1826 ]
1827 ]
1828 ]
1828 ]
1829 ]
1829 ]
1830 ]
1831 ]
1832 ]
1833 ]
1834 ]
1835 ]
1836 ]
1837 ]
1838 ]
1838 ]
1839 ]
1839 ]
1840 ]
1841 ]
1842 ]
1843 ]
1844 ]
1845 ]
1846 ]
1847 ]
1848 ]
1848 ]
1849 ]
1849 ]
1850 ]
1851 ]
1852 ]
1853 ]
1854 ]
1855 ]
1856 ]
1857 ]
1858 ]
1858 ]
1859 ]
1859 ]
1860 ]
1861 ]
1862 ]
1863 ]
1864 ]
1865 ]
1866 ]
1867 ]
1868 ]
1868 ]
1869 ]
1869 ]
1870 ]
1871 ]
1872 ]
1873 ]
1874 ]
1875 ]
1876 ]
1877 ]
1877 ]
1878 ]
1878 ]
1879 ]
1879 ]
1880 ]
1881 ]
1882 ]
1883 ]
1884 ]
1885 ]
1886 ]
1887 ]
1888 ]
1888 ]
1889 ]
1889 ]
1890 ]
1891 ]
1892 ]
1893 ]
1894 ]
1895 ]
1896 ]
1897 ]
1898 ]
1898 ]
1899 ]
1899 ]
1900 ]
1901 ]
1902 ]
1903 ]
1904 ]
1905 ]
1906 ]
1907 ]
1908 ]
1909 ]
1909 ]
1910 ]
1911 ]
1912 ]
1913 ]
1914 ]
1915 ]
1916 ]
1917 ]
1918 ]
1918 ]
1919 ]
1919 ]
1920 ]
1921 ]
1922 ]
1923 ]
1924 ]
1925 ]
1926 ]
1927 ]
1928 ]
1928 ]
1929 ]
1929 ]
1930 ]
1931 ]
1932 ]
1933 ]
1934 ]
1935 ]
1936 ]
1937 ]
1938 ]
1938 ]
1939 ]
1939 ]
1940 ]
1941 ]
1942 ]
1943 ]
1944 ]
1945 ]
1946 ]
1947 ]
1948 ]
1948 ]
1949 ]
1949 ]
1950 ]
1951 ]
1952 ]
1953 ]
1954 ]
1955 ]
1956 ]
1957 ]
1958 ]
1958 ]
1959 ]
1959 ]
1960 ]
1961 ]
1962 ]
1963 ]
1964 ]
1965 ]
1966 ]
1967 ]
1968 ]
1968 ]
1969 ]
1969 ]
1970 ]
1971 ]
1972 ]
1973 ]
1974 ]
1975 ]
1976 ]
1977 ]
1978 ]
1978 ]
1979 ]
1979 ]
1980 ]
1981 ]
1982 ]
1983 ]
1984 ]
1985 ]
1986 ]
1987 ]
1988 ]
1988 ]
1989 ]
1989 ]
1990 ]
1991 ]
1992 ]
1993 ]
1994 ]
1995 ]
1996 ]
1997 ]
1
```

## 9- Insertar un curso:



The screenshot shows the Postman interface with a 'POST Insertar un curso' request. The URL is `localhost:8080/api/v1/golearnix/courses`. The Body tab contains the following JSON payload:

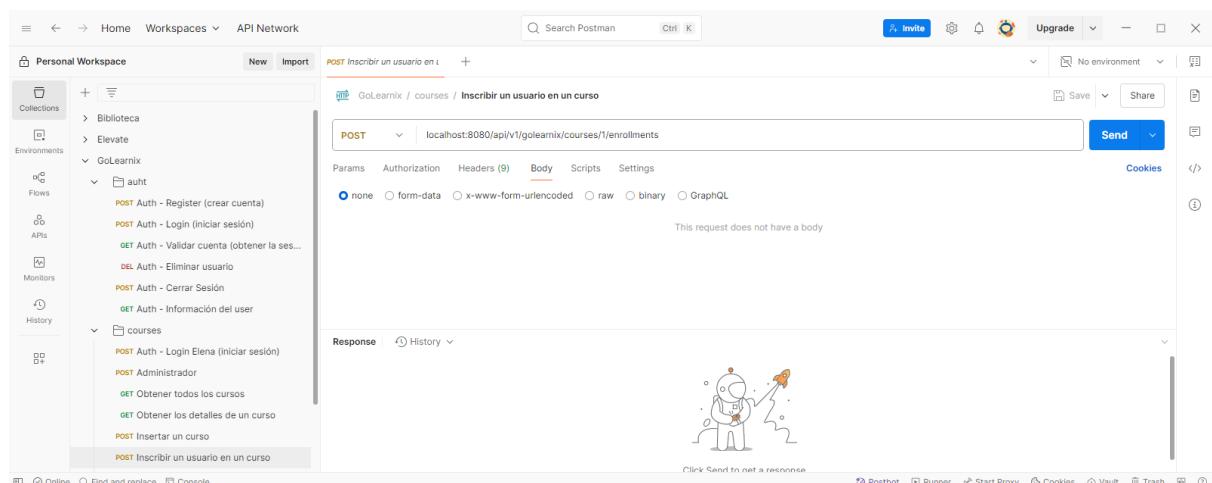
```

1 {
2     "title": "Introducción a la Programación",
3     "description": "Curso básico para aprender los fundamentos de la programación.",
4     "instructor": [
5         {"id": "1a2b3c4d-5e6f-7a8b-9c0d-1e2f3a4b5c6d"}
6     ],
7     "category": [
8         {"id": 1}
9     ],
10    "sections": [
11        {"id": 1},
12        {"id": 2}
13    ],
14    "reviews": [
15        {"id": 1},
16        {"id": 2}
17    ],
18    "enrollments": [
19        {"id": 1},
20        {"id": 2}
21    ]
22 }

```

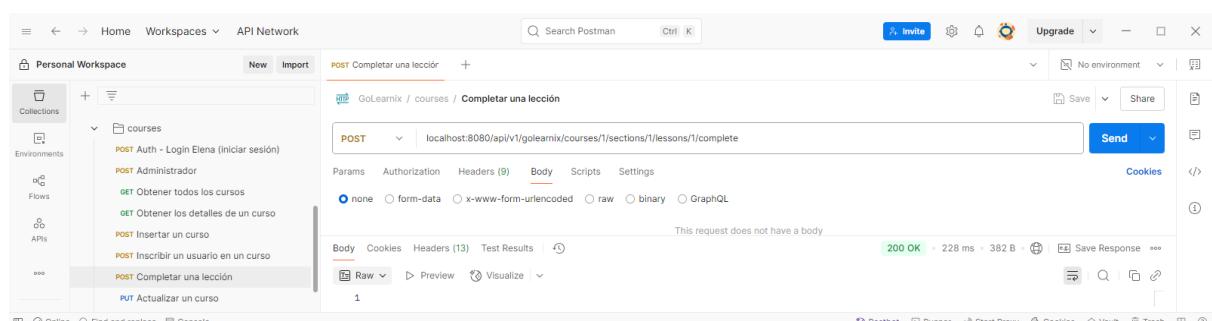
The response status is 200 OK, with a response body of '1'.

## 10- Inscribir un usuario en un curso:



The screenshot shows the Postman interface with a 'POST Inscribir un usuario en un curso' request. The URL is `localhost:8080/api/v1/golearnix/courses/1/enrollments`. The Headers tab shows 'Content-Type: application/json'. The response status is 200 OK, with a response body of '1'.

## 11- Completar una lección:



The screenshot shows the Postman interface with a 'POST Completar una lección' request. The URL is `localhost:8080/api/v1/golearnix/courses/1/sections/1/lessons/1/complete`. The Headers tab shows 'Content-Type: application/json'. The response status is 200 OK, with a response body of '1'.



## 12- Actualizar un curso:

The screenshot shows the Postman interface with a PUT request to `localhost:8080/api/v1/golearnix/courses/3`. The request body is a JSON object:

```

1  {
2    "title": "Introducción a la Programación",
3    "description": "Curso básico para aprender los fundamentos de la programación.",
4    "instructor": [
5      {
6        "id": "1a2b3c4d-5e6f-7a8b-9c0d-1e2f3a4b5c6d"
7      }
8    ],
9    "category": {
10      "id": 1
11    }
12  }

```

## 13- Eliminar un curso:

The screenshot shows the Postman interface with a DELETE request to `localhost:8080/api/v1/golearnix/courses/1`. The response status is 200 OK.

## 14- Eliminar un usuario:

The screenshot shows the Postman interface with a DELETE request to `localhost:2003/api/v1/user/delete`. The response status is 200 OK and the response body is:

```

1  {
2    "success": true,
3    "message": "Usuario eliminado exitosamente"
4  }

```



Además, se puede observar el mensaje en la cola de RabbitMQ:

The screenshot shows the RabbitMQ Management Interface with the 'Queues and Streams' tab selected. A single message is present in the 'golearnix.events' queue. The message details are as follows:

- Exchange:** golearnix.events
- Routing Key:** user.deleted
- Redelivered:** 0
- Properties:** content\_type: application/json
- Payload:** {"event": "UserDeleted", "data": {"user\_id": "87c493cb-e654-4843-8d93-42e2fb4b1e9e", "email": "juan@example.com"}}, 188 bytes
- Encoding:** string

Para finalizar, los consumidores que tiene en la sección de abajo *Consumers*:

The screenshot shows the RabbitMQ Management Interface with the 'Consumers' section selected. It displays the following consumer details:

- Features:** x-dead-letter-exchange: golearnix.events.dlx, x-dead-letter-routing-key: user.deleted.dlq, durable: true, queue storage version: 1
- State:** idle
- Consumers:** 1
- Consumer capacity:** 97%
- Messages:** Total 2, Ready 0, Unacked 2, In memory 2, Persistent 0, Transient 0, Paged Out 0
- Message body bytes:** 216 B, Process memory: 14 KIB

Below this, a table lists the consumer configuration:

Channel	Consumer tag	Ack required	Exclusive	Prefetch count	Active	Activity status	Consumer Timeout	Arguments
172.19.0.1:56794 (1)	amq.ctag-y77bj6LdLTlgBVU87fxoqg	*	o	250	*	up	1800000	



## 7. PROPUESTAS DE MEJORA

Con el objetivo de aumentar la robustez, la escalabilidad y la mantenibilidad de la plataforma GoLearnix, se plantean a continuación distintas iniciativas de mejora, detalladas de forma exhaustiva y organizadas por áreas de actuación:

### A) Clústeres de alta disponibilidad

La puesta en producción de sistemas críticos exige una tolerancia a fallos continua. Para ello, una buena práctica es desplegar PostgreSQL, Redis y el sistema de mensajería en clústeres configurados con réplicas maestras y esclavas, replicación síncrona o asíncrona según el caso, y mecanismos automáticos de comutación por error (failover). En el caso de PostgreSQL, soluciones como Patroni o PgPool-II garantizarían la disponibilidad del almacenamiento relacional; para Redis, Sentinel o Redis Cluster permiten distribuir la carga y detectar caídas de nodos; y en RabbitMQ, la configuración de clúster nativo o la adopción de un broker alternativo con particionamiento aseguran la continuidad de la mensajería.

### B) Descomposición en microservicios de la gestión de cursos

Aunque actualmente la lógica de cursos, inscripciones y seguimiento de progresos reside en un único servicio, la separación de estos dominios en microservicios independientes aportaría beneficios significativos. Cada servicio podría escalar de forma autónoma, adoptar la tecnología más adecuada (por ejemplo, un servicio de inscripciones basado en eventos puros) y permitir despliegues y actualizaciones desacoplados. Asimismo, facilitaría el trabajo en equipos especializados, reduciría el blast radius de cambios y mejoraría la seguridad al aplicar políticas de acceso específicas a cada microservicio.

### C) Migración de eventos a Kafka con Debezium

El patrón de mensajería actual basado en RabbitMQ puede evolucionar hacia una arquitectura de streaming con Apache Kafka. La integración de Debezium para Change Data Capture sobre PostgreSQL posibilitaría la generación automática de eventos basados en modificaciones del esquema relacional, sin necesidad de instrumentar manualmente cada operación. De este modo, todos los cambios en las entidades quedarían registradas en topics de Kafka, garantizando un historial inmutable, escalabilidad horizontal de consumo y la capacidad de reprocesar flujos de datos desde cualquier punto temporal.



#### D) Pruebas unitarias

Para asegurar la calidad del código y minimizar regresiones, se recomienda aumentar la cobertura de tests unitarios en todas las capas: servicios de aplicación, ensambladores (Assembler), validadores y adaptadores de entrada/salida. El uso de JUnit 5 junto con Mockito permitirá aislar dependencias, simular repositorios y verificar comportamientos en cada escenario.

#### E) Pruebas de integración

Más allá de las pruebas unitarias, las pruebas de integración end-to-end replican el entorno real de producción. Herramientas como Testcontainers pueden levantar contenedores efímeros de PostgreSQL, Redis y RabbitMQ/Kafka para ejecutar suites de pruebas que validen la interacción completa entre Spring Boot, la base de datos relacional, el almacén en caché y el broker de mensajes. Estas pruebas confirman que los flujos de negocio, las transacciones y la replicación de datos funcionan como se espera, y detectan posibles problemas de configuración o de compatibilidad de versiones.

Adoptar progresivamente estas propuestas permitirá elevar la calidad arquitectónica, garantizar la continuidad del servicio y acelerar la evolución del producto mínimo viable.



## 8. VALORACIÓN PERSONAL

El desarrollo de este Trabajo de Fin de Grado ha representado un desafío de elevada complejidad técnica y organizativa, en el que la constancia y el rigor han resultado fundamentales para la consecución de los objetivos planteados. La definición de una arquitectura robusta, la selección de herramientas adecuadas y la implementación de patrones de diseño han permitido afrontar con éxito los múltiples aspectos que caracterizan a los sistemas distribuidos modernos.

La investigación y aplicación de conceptos como la Arquitectura Hexagonal, el Domain-Driven Design o el patrón Saga han aportado claridad y cohesión al conjunto de microservicios. El proceso de configurar entornos, desde la orquestación de contenedores con Docker hasta la conexión segura de Spring Boot con PostgreSQL, Redis y RabbitMQ, ha exigido un nivel detallado de coordinación, que ha reforzado la capacidad de diseñar flujos de despliegue reproducibles y tolerantes a fallos. Gracias a esta experiencia, las habilidades en automatización de migraciones con Flyway, generación de contratos OpenAPI y externalización de la configuración han alcanzado un grado de madurez profesional.

La adopción de mecanismos de seguridad basados en JWT, jerarquías de roles y filtros declarativos ha garantizado que la plataforma cumpla con los estándares de integridad y confidencialidad. Asimismo, la implementación de pruebas funcionales mediante Postman y la estructuración de colecciones automatizadas han facilitado la validación continua de los servicios, subrayando la importancia de la calidad y la trazabilidad en cada iteración.

La elaboración de este proyecto ha servido para consolidar la práctica de generar documentación exhaustiva y actualizada, garantizando que cada aspecto, desde la configuración del entorno hasta el comportamiento de los endpoints, quede recogido de forma precisa y accesible.

Como resultado, el proyecto se ha convertido en un ejercicio integral en el que la teoría se ha aplicado de forma práctica, reforzando la capacidad de resolver retos técnicos complejos y de estructurar soluciones sostenibles. Los conocimientos y metodologías puestas en práctica, arquitecturas limpias, gestión de eventos, externalización de configuración y patrones de diseño, configuran una base sólida para futuros desarrollos, orientados siempre a la calidad, la escalabilidad y la mantenibilidad.



## 9. PUNTOS A DESTACAR

En el desarrollo se ha apostado por la Arquitectura Hexagonal, que aísla el núcleo de dominio de las capas externas mediante la definición de puertos (interfaces) y adaptadores. Este enfoque garantiza que la lógica de negocio permanezca libre de dependencias tecnológicas, de modo que la introducción de nuevos canales de entrada, como APIs REST o consumidores de eventos, o de nuevos mecanismos de persistencia, como Redis o PostgreSQL, pueda realizarse sin modificar el corazón de la aplicación.

El uso de Domain-Driven Design ha proporcionado un lenguaje común y una estructura de módulos alineada con los conceptos reales del negocio. Entidades, agregados y servicios de dominio se han organizado en torno a los procesos esenciales de la plataforma, lo que facilita la comprensión, el mantenimiento y la evolución de las reglas de negocio.

Para asegurar la consistencia eventual entre las bases de datos relacional y de cache, se ha implementado un patrón Saga basado en eventos de dominio. Cada cambio relevante dispara un evento interno, cuyos listeners replican la acción en Redis o, en caso de fallo, aplican una operación compensatoria en PostgreSQL. De este modo, se evita la complejidad de las transacciones distribuidas y se mantiene la resiliencia del sistema ante errores parciales.

La abstracción del acceso a datos se ha materializado en repositorios que actúan como puertos de salida. Estos repositorios definen contratos estables que los servicios de dominio consumen sin conocer las implementaciones subyacentes, ya sean JPA, Redis OM Spring u otras tecnologías. Esta capa de abstracción permite reemplazar o actualizar los mecanismos de persistencia sin impacto en la lógica central.

La centralización de la lógica de mapeo en componentes Assembler ha reducido la duplicación de código y ha respetado el principio de responsabilidad única. Todo el trabajo de transformación de objetos, especialmente en operaciones de inserción y actualización con relaciones sencillas, queda confinado a estos ensambladores especializados.

La configuración de la aplicación se ha externalizado por completo en un ficheros de variables de entorno. Esta práctica elimina hard-codes y facilita la parametrización de los distintos entornos (desarrollo, staging, producción), garantizando un despliegue coherente y seguro.

La generación automática del contrato OpenAPI a partir de las anotaciones de los controladores ha acelerado el proceso de documentación y facilitado la generación de clientes HTTP. Springdoc OpenAPI ha convertido en simples anotaciones todo el trabajo de definir rutas, parámetros y posibles respuestas, integrándose de manera transparente en los pipelines de integración continua.

Finalmente, la gestión de versiones del esquema relacional con Flyway ha proporcionado un control exhaustivo de las migraciones. La definición de una línea de base, la validación previa de cambios y la organización sistemática de los scripts de migración aseguran que cada despliegue actualice la base de datos de forma ordenada, auditável y reversible en caso de error.



## 10. CONCLUSIÓN

A lo largo de este proyecto se ha demostrado la relevancia de adoptar arquitecturas modulares, patrones de diseño sólidos y prácticas de configuración externalizada para el desarrollo de sistemas distribuidos. La aplicación simultánea de la Arquitectura Hexagonal, el Domain-Driven Design y el patrón Saga ha permitido obtener una solución coherente, que aísla el núcleo de dominio de las dependencias tecnológicas y garantiza la consistencia eventual entre múltiples almacenes de datos. La generación automatizada de contratos OpenAPI y el uso de Flyway para el versionado de esquemas han reforzado la trazabilidad y la reproducibilidad de los despliegues, dos factores críticos en entornos profesionales.

Del mismo modo, la integración de mecanismos de seguridad basados en JWT, roles jerárquicos y filtros declarativos ha proporcionado un marco riguroso de control de acceso, mientras que la externalización de la configuración en un único fichero de variables de entorno ha facilitado la parametrización en distintos escenarios (desarrollo, staging y producción). La contenedorización con Docker y la orquestación de servicios han puesto de manifiesto la importancia de diseñar flujos de despliegue reproducibles, capaces de adaptarse con agilidad a cambios en la demanda.

Las propuestas de mejora planteadas, clústeres de alta disponibilidad, descomposición en microservicios, migración a Kafka con Debezium, pruebas unitarias e integración y despliegue automatizado, refuerzan el compromiso con la calidad, la escalabilidad y la resiliencia. Cada una de estas líneas de trabajo se fundamenta en la necesidad de ofrecer servicios ininterrumpidos y evolutivos, capaces de responder con rapidez a situaciones adversas y de anticiparse a las demandas del mercado.

En definitiva, este proyecto ha servido como ejercicio integrado de investigación, diseño e implementación, en el que la combinación de teoría y práctica ha brindado una visión completa de las exigencias y soluciones asociadas a los sistemas distribuidos modernos. La experiencia adquirida y los patrones aplicados sientan las bases para futuros desarrollos, orientados siempre a la excelencia arquitectónica, la mantenibilidad y la alta disponibilidad de las plataformas de software.



## 11. BIBLIOGRAFÍA

Academia.edu. (n.d.). *Bases de datos II distribuidas: Procesos de transacciones* [Documento en línea].

[https://www.academia.edu/36909925/BASES DE DATOS II DISTRIBUIDAS Procesos de Transacciones](https://www.academia.edu/36909925/BASES_DE_DATOS_II_DISTRIBUIDAS_Procesos_de_Transacciones)

Alldevstack. (n.d.-a). *Go Fiber tutorial: Go Fiber routing*.

<https://www.alldevstack.com/es/go-fiber-tutorial/go-fiber-routing.html>

Alldevstack. (n.d.-b). *Go Fiber tutorial: PProf performance analysis*.

<https://www.alldevstack.com/es/go-fiber-tutorial/go-fiber-pprof-performance-analysis.html>

Amazon Web Services. (n.d.). *Saga pattern*.

<https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/saga-pattern.html>

Ankur. (2019, abril). *Saga pattern in microservices*. Medium.

<https://ankur-javaarch.medium.com/saga-pattern-in-microservices-11a23204d86c>

Apiumhub. (n.d.-a). *Sacrificial architecture*.

<https://apiumhub.com/tech-blog-barcelona/sacrificial-architecture/>

Apiumhub. (n.d.-b). *Arquitectura de sacrificio*

<https://apiumhub.com/es/tech-blog-barcelona/arquitectura-de-sacrificio/>

Azure Microsoft. (n.d.). *What is Java Spring Boot?* Cloud Computing Dictionary.

<https://azure.microsoft.com/es-es/resources/cloud-computing-dictionary/what-is-java-spring-boot/>

Baeldung. (n.d.). *Java DTO pattern*.

<https://www.baeldung.com/java-dto-pattern>

Block & Capital. (n.d.). *Integrando principios SOLID*

<https://blockandcapital.com/integrando-principios-solid/>

Byron Vargas. (n.d.). *¿Qué es la arquitectura de tres niveles?*

<https://www.byronvargas.com/web/que-es-la-arquitectura-de-tres-niveles/>

Codd, E. F. (1970). *A relational model of data for large shared data banks*. Communications of the ACM, 13(6), 377–387.



Confluent. (n.d.). *Event-driven architecture*.

<https://www.confluent.io/es-es/learn/event-driven-architecture>

DevelopN. (n.d.). *Software architecture patterns: Monolithic architecture*.

<https://www.developn.net/en/article/software-architecture-patterns-monolithic-architecture>

DotNetTutorials. (n.d.). *Repository design pattern in Java*.

<https://dotnettutorials.net/lesson/repository-design-pattern-in-java/>

EDF ResearchGate. (n.d.). García-Molina, H., & Salem, K. *Domain-Driven Design tackling complexity in the heart of business software* [Artículo].

[https://www.researchgate.net/publication/2563768\\_Domain\\_Driven\\_Design\\_Tackling\\_Complexity\\_in\\_the\\_Heart\\_of\\_Business\\_Software](https://www.researchgate.net/publication/2563768_Domain_Driven_Design_Tackling_Complexity_in_the_Heart_of_Business_Software)

FreeCodeCamp. (2021, abril). *Los principios SOLID explicados en español*.

<https://www.freecodecamp.org/espanol/news/los-principios-solid-explicados-en-espanol/>

Gist: CGavrila. (n.d.). *Clean Architecture overview* [Gist].

<https://gist.github.com/CGavrila/7a41e812e4c35e2e15c505db922a7924>

Gyata.ai. (n.d.). *Golang Fiber*.

<https://www.gyata.ai/es/golang/golang-fiber>

IBM. (n.d.). *Three-tier architecture*.

<https://www.ibm.com/mx-es/topics/three-tier-architecture>

Innowise. (n.d.). *Desarrollo Java Spring Boot*.

<https://innowise.com/es/tecnologias/desarrollo-java-spring-boot/>

Interempresas. (n.d.). *La modularidad como filosofía de construcción industrializada*.

<https://www.interempresas.net/construccion-industrializada/Articulos/351982-La-Modularidad-como-filosofia-de-construccion-industrializada.html>

JavierMartinalonso. (2017, 2 de enero). *Spring Boot*.

<https://javiermartinalonso.github.io/spring/2017/01/02/spring-boot.html>

Jorgesanchez. (n.d.). *Modelo relacional*.

<https://jorgesanchez.net/manuales/gbd/modelo-relacional.html>



Jordan Fawcett (Syracuse University). (n.d.). *Lecture20: References: Thesis-Magnus* [Handout]. Recuperado 16 de mayo de 2025, de <https://ecs.syr.edu/faculty/fawcett/Handouts/cse776/Lecture20/References/Thesis-Magnus.htm>

Langlade, A. (2024). *An Introduction to the Repository pattern*. Recuperado 16 de mayo de 2025, de <https://www.arnaudlanglade.com/repository-design-pattern/>

LinkedIn – Nina Osipova. (n.d.). *Arquitecturas de software: Hexagonal, onion y en Java*. Recuperado 16 de mayo de 2025, de <https://es.linkedin.com/pulse/arquitecturas-de-software-hexagonal-onion-y-en-java-nina-osipova-aohof>

LinkedIn – Ulcof. (n.d.). *Los principios SOLID, pilares fundamentales del desarrollo de software*. Recuperado 16 de mayo de 2025, de <https://es.linkedin.com/pulse/los-principios-solid-pilares-fundamentales-del-desarrollo-de-software-ulcof>

Lifewire. (n.d.). *NoSQL: An overview of NoSQL databases*.  
<https://www.lifewire.com/nosql-an-overview-of-nosql-databases-2495393>

Likonet. (2024, 12 de septiembre). *Conceptos teóricos de programación: Arquitectura hexagonal*.  
<https://likonet.es/2024/09/12/conceptos-teoricos-de-programacion-arquitectura-hexagonal/>

Martin Fowler. (n.d.-a). *Domain-Driven Design*.  
<https://www.martinfowler.com/tags/domain%20driven%20design.html>

Martin Fowler. (n.d.-b). *Sacrificial Architecture*.  
<https://martinfowler.com/bliki/SacrificialArchitecture.html>

Martin Fowler (EAA catalog). (n.d.). *Data Transfer Object*.  
<https://www.martinfowler.com/eaaCatalog/dataTransferObject.html>

Mentorestech. (n.d.-a). *Arquitectura limpia (Clean Architecture)*.  
<https://mentorestech.com/resource-blog-content/arquitectura-limpia-clean-architecture>

Mentorestech. (n.d.-b). *¿Qué es Swagger?*  
<https://www.mentorestech.com/resource-guide/qa/question/que-es-swagger>

Microsoft Azure. (n.d.). *Cloud Computing Dictionary*.  
<https://azure.microsoft.com/es-es/resources/cloud-computing-dictionary/what-is-javascript-boot/>



OpenWebinars. (n.d.). *Desarrollo web con el framework Fiber (Go)*.

<https://openwebinars.net/cursos/desarrollo-web-framework-fiber-go/>

O'Reilly. (2021). *Spring Boot en práctica* (Ch. 1).

<https://www.oreilly.com/library/view/spring-boot-en/9781098188016/ch01.html>

Paradigma Digital. (n.d.). *Patrones de arquitectura de microservicios: Qué son y ventajas*.

<https://www.paradigmadigital.com/dev/patrones-arquitectura-microservicios-que-son-ventajas/>

Paraninfo. (n.d.). *UF2175—Diseño de bases de datos relacionales*.

<https://www.paraninfo.es/catalogo/9788428363914/uf2175---diseno-de-bases-de-datos-relacionales>

PercoGuru. (n.d.). *Getting started with APIs in Golang feat. Fiber and GORM*.

<https://dev.to/percoguru/getting-started-with-apis-in-golang-feat-fiber-and-gorm-2n34>

Peerdh. (n.d.). *Comparative analysis of middleware frameworks in Go for web server scalability*.

<https://peerdh.com/es/blogs/programming-insights/comparative-analysis-of-middleware-frameworks-in-go-for-web-server-scalability>

Pragma. (n.d.). *¿Qué es un microservicio? Conceptos clave y ejemplos*.

<https://www.pragma.co/es/blog/que-es-un-microservicio-conceptos-claves-y-ejemplos>

Red Hat. (n.d.). *¿Qué son los microservicios?*

<https://www.redhat.com/es/topics/microservices/what-are-microservices>

ResearchGate. (2017). *Overview of a Domain-Driven Design approach to build microservice-based applications*.

[https://www.researchgate.net/publication/316492773\\_Overview\\_of\\_a\\_Domain-Driven\\_Design\\_Approach\\_to\\_Build\\_Microservice-Based\\_Applications](https://www.researchgate.net/publication/316492773_Overview_of_a_Domain-Driven_Design_Approach_to_Build_Microservice-Based_Applications)

Santander Open Academy. (n.d.). *Arquitectura hexagonal*.

<https://www.santanderopenacademy.com/es/blog/arquitectura-hexagonal.html>

Scribd. (n.d.). *Arquitectura y patrones* [Documento en línea].

<https://es.scribd.com/document/49708406/arquitectura-y-patrones>

SoftEngBook.org. (n.d.). *Domain-Driven Design*.

<https://softengbook.org/articles/ddd>



StackOverflow Blog. (2021, marzo 1). *Sacrificial architecture: Learning from abandoned systems.*

<https://stackoverflow.blog/2021/03/01/sacrificial-architecture-learning-from-abandoned-systems>

StackScale. (n.d.). *Bases de datos NoSQL.*

<https://www.stackscale.com/es/blog/bases-de-datos-nosql/>

Thomas, T. (n.d.). *Saga pattern and microservices architecture.* Medium.

[https://medium.com/@tomasz\\_96685/saga-pattern-and-microservices-architecture-d4b46071afcf](https://medium.com/@tomasz_96685/saga-pattern-and-microservices-architecture-d4b46071afcf)

TutorialQ. (n.d.). *Unleashing the power of event-driven architecture: Key concepts.*

<https://tutorialq.medium.com/unleashing-the-power-of-event-driven-architecture-key-concepts-15fe3ccc5061>

UpM – Repositorio UPM. (n.d.). *Autor desconocido. Título del recurso* [Tesis]. Universidad Politécnica de

Madrid. <https://oa.upm.es/64213/>

Wolf Agencia Marketing. (n.d.). *¿Qué es la arquitectura de 3 capas?*

<https://wolfagenciademarketing.com/que-es-la-arquitectura-de-3-capas/>

Wikipedia contributors. (n.d.-a). *Arquitectura de microservicios.* Wikipedia.

[https://es.wikipedia.org/wiki/Arquitectura\\_de\\_microservicios](https://es.wikipedia.org/wiki/Arquitectura_de_microservicios)

Wikipedia contributors. (n.d.-b). *Arquitectura de tres capas.* Wikipedia.

<https://es.wikipedia.org/wiki/Presentaci%C3%B3n%20abstracci%C3%B3n%20de%20control>

Wikipedia contributors. (n.d.-c). *Arquitectura hexagonal.* Wikipedia.

[https://es.wikipedia.org/wiki/Arquitectura\\_hexagonal](https://es.wikipedia.org/wiki/Arquitectura_hexagonal)

Wikipedia contributors. (n.d.-d). *Diseño guiado por el dominio.* Wikipedia.

[https://es.wikipedia.org/wiki/Domain-Driven\\_Design](https://es.wikipedia.org/wiki/Domain-Driven_Design)

Wikipedia contributors. (n.d.-e). *Ingeniería de software.* Wikipedia.

[https://es.wikipedia.org/wiki/Ingenier%C3%ADa\\_de\\_software](https://es.wikipedia.org/wiki/Ingenier%C3%ADa_de_software)

Wikipedia contributors. (n.d.-f). *Modularidad (informática).* Wikipedia.

[https://es.wikipedia.org/wiki/Modularidad\\_\(inform%C3%A1tica\)](https://es.wikipedia.org/wiki/Modularidad_(inform%C3%A1tica))



Wikipedia contributors. (n.d.-g). *Patrón de diseño*. Wikipedia.

[https://es.wikipedia.org/wiki/Patr%C3%B3n\\_de\\_dise%C3%BAo](https://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%BAo)

Wikipedia contributors. (n.d.-h). *Presentación–abstracción–control*. Wikipedia.

<https://es.wikipedia.org/wiki/Presentaci%C3%B3n%20abstracci%C3%B3n%20control>

Wikipedia contributors. (n.d.-i). *RAML (lenguaje)*. Wikipedia.

[https://es.wikipedia.org/wiki/RAML\\_%28lenguaje%29](https://es.wikipedia.org/wiki/RAML_%28lenguaje%29)

Wikipedia contributors. (n.d.-j). *Síntesis de las reglas de Codd*. Wikipedia.

[https://es.wikipedia.org/wiki/12\\_reglas\\_de\\_Codd](https://es.wikipedia.org/wiki/12_reglas_de_Codd)

Wikipedia contributors. (n.d.-k). *Normalización de bases de datos*. Wikipedia.

[https://es.wikipedia.org/wiki/Normalizaci%C3%B3n\\_de\\_bases\\_de\\_datos](https://es.wikipedia.org/wiki/Normalizaci%C3%B3n_de_bases_de_datos)

Wikipedia contributors. (n.d.-l). *Edgar Frank Codd*. Wikipedia.

[https://es.wikipedia.org/wiki/Edgar\\_Frank\\_Codd](https://es.wikipedia.org/wiki/Edgar_Frank_Codd)

Wikipedia contributors. (n.d.-m). *ACID*. Wikipedia.

<https://es.wikipedia.org/wiki/ACID>

