UNIVERSIDAD TECNOLOGICA DE TEHUACAN

INGENIERIA EN DESARROLLO Y GESTION DE SOFTWARE

ACTIVIDAD 2: MODELADO DE ARQUITECTURAS DE SOFTWARE (C4 + UML)

INTEGRANTES:

ARIEL ABIMAEL CHACÓN HERRERA

GERMAN YAIR MARTINEZ BOLAÑOS

CARLOS ANDRES ARRIAGA MARQUEZ

ENRIQUE JULIÁN GRACIA LÓPEZ

21/09/2025

SEPTIEMBRE-DICIEMBRE

Asignatura: Arquitectura de Software

JOSÉ MIGUEL CARRERA PACHECO

Actividad Semana 3: Modelado de Arquitecturas de Software (C4 + UML)

Contexto del caso

La universidad quiere rediseñar su sistema de **gestión de inscripciones y pagos en línea**. Actualmente es un sistema monolítico que presenta problemas de **escalabilidad, disponibilidad y mantenibilidad**. Se requiere evolucionarlo a una **arquitectura SaaS escalable**, con microservicios y micro frontends.

Requerimientos Funcionales:

- **Gestión de Cuentas:** Los alumnos deben poder registrarse, iniciar sesión y gestionar su perfil.
- **Consulta Académica:** Los alumnos deben poder ver la oferta de cursos, detalles y horarios.
- **Proceso de Inscripción:** Los alumnos deben poder seleccionar y añadir cursos a un "carrito" y confirmar su inscripción.
- **Gestión de Pagos:** El sistema debe permitir a los alumnos pagar las tasas de inscripción utilizando una pasarela de pagos externa.
- **Administración:** El personal administrativo debe poder gestionar cursos, ver las inscripciones de los alumnos y supervisar los estados de los pagos.

Requerimientos No Funcionales:

- **Escalabilidad:** El sistema debe manejar picos de demanda durante los períodos de inscripción sin degradar el rendimiento.
- **Disponibilidad:** El sistema debe tener una alta disponibilidad (ej. 99.9%), minimizando el tiempo de inactividad.
- **Mantenibilidad:** Las actualizaciones o la corrección de errores en un módulo (ej. pagos) no deben requerir el redespliegue de todo el sistema.
- **Seguridad:** La información personal y de pago de los usuarios debe estar protegida en todo momento.

Diagrama de Contexto (Nivel 1)

Este Diagrama de Contexto ofrece la visión más amplia y simplificada del sistema, mostrándolo como una "caja negra". Su propósito es identificar a los usuarios principales, como el **Alumno** y el **Administrativo**, y las interacciones clave con sistemas externos de los que dependemos, como la **Pasarela de Pagos** y el **Sistema de Notificaciones**. Este diagrama es fundamental para entender el ecosistema en el que vive nuestra aplicación sin entrar en detalles técnicos.

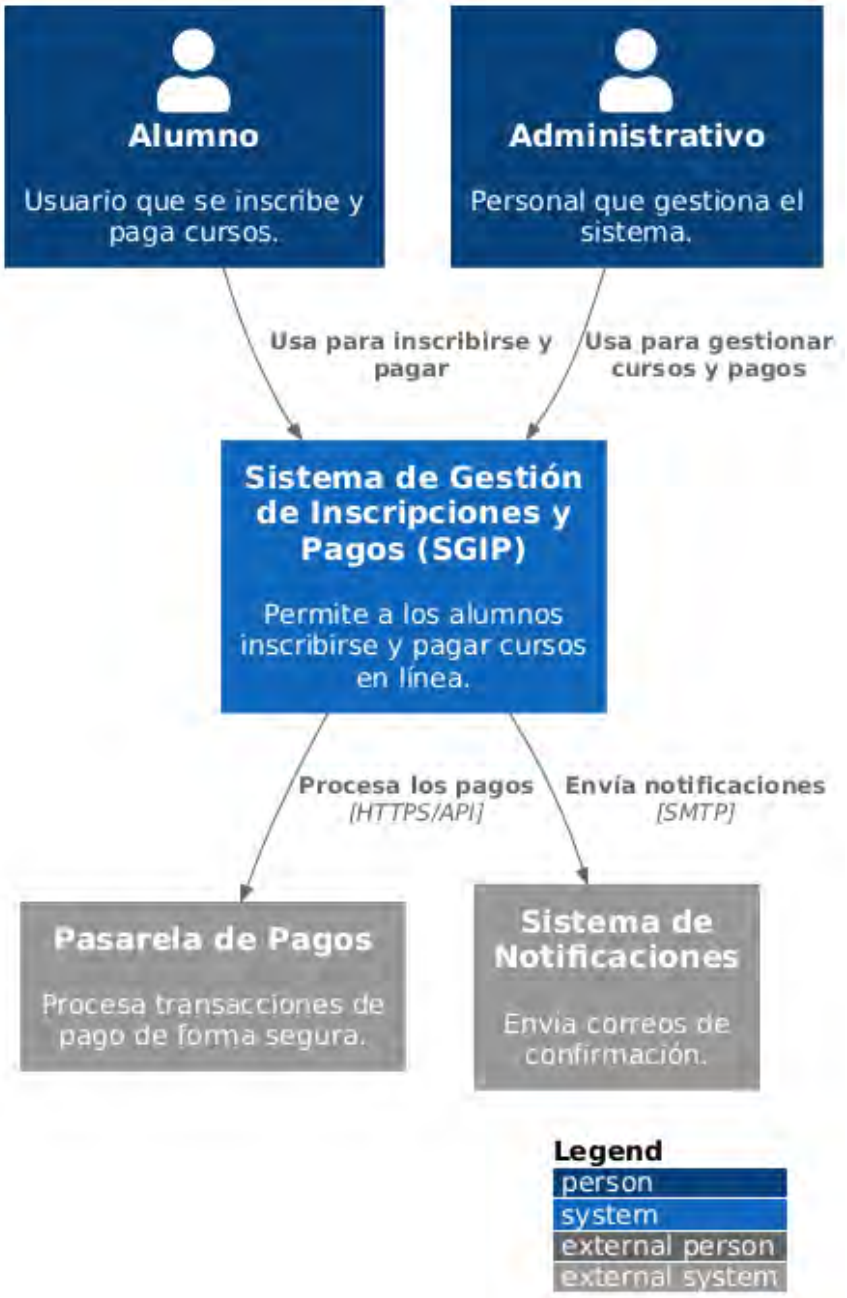


Diagrama de Contenedores (Nivel 2)

Al hacer "zoom in" en el sistema, este Diagrama de Contenedores descompone la arquitectura en sus principales bloques tecnológicos ejecutables. Muestra cómo la **Aplicación Web (SPA)** se comunica a través de un **API Gateway** centralizado con los diferentes microservicios de backend (`Autenticación`, `Inscripciones` y `Pagos`).

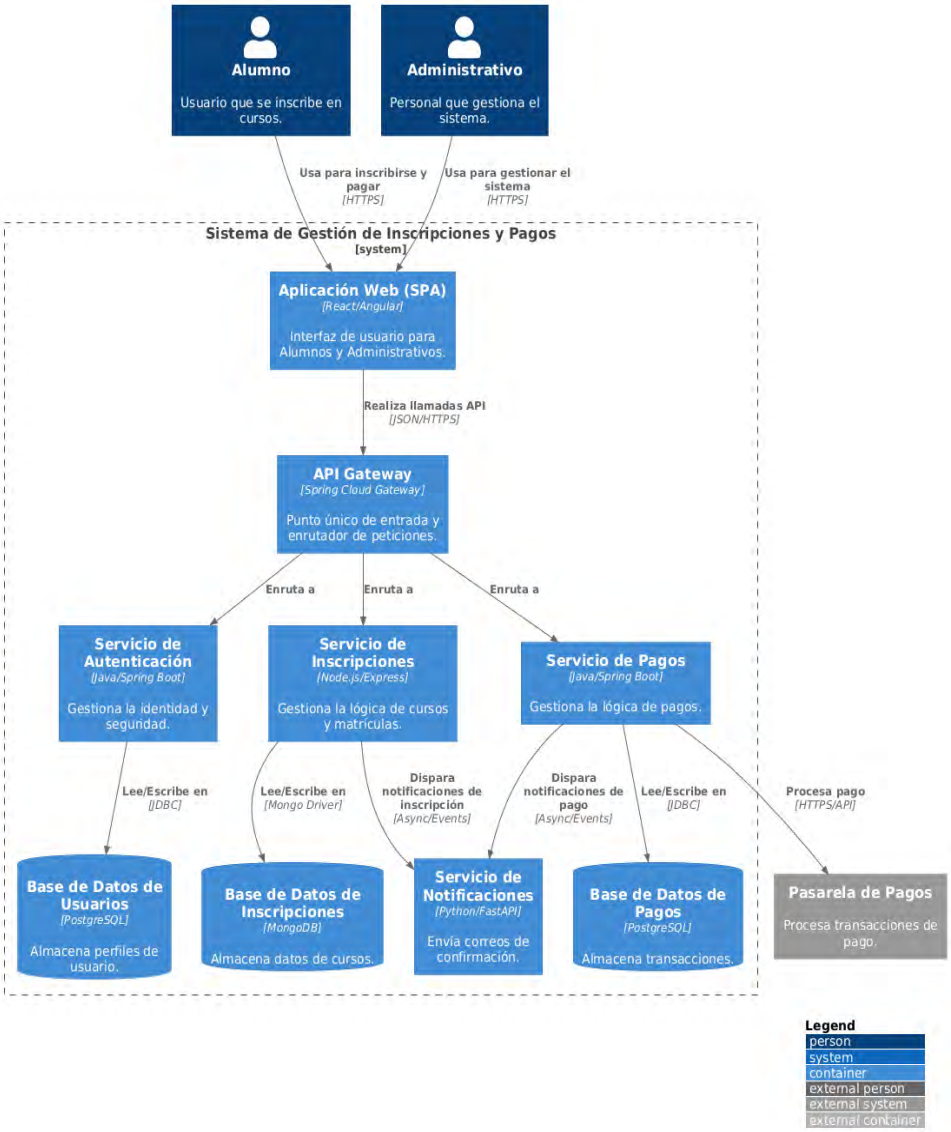


Diagrama de Componentes (Nivel 3)

Este diagrama ofrece una vista detallada del interior de un solo contenedor: el **Servicio de Pagos**. Su objetivo es mostrar cómo se distribuyen las responsabilidades entre sus componentes internos. Se puede observar un patrón claro donde el `Controller` gestiona las peticiones, el `Service` orquesta la lógica de negocio, el `Adapter` se comunica con sistemas externos y el `Repository` abstrae el acceso a los datos. Este nivel es clave para entender el diseño interno de un microservicio

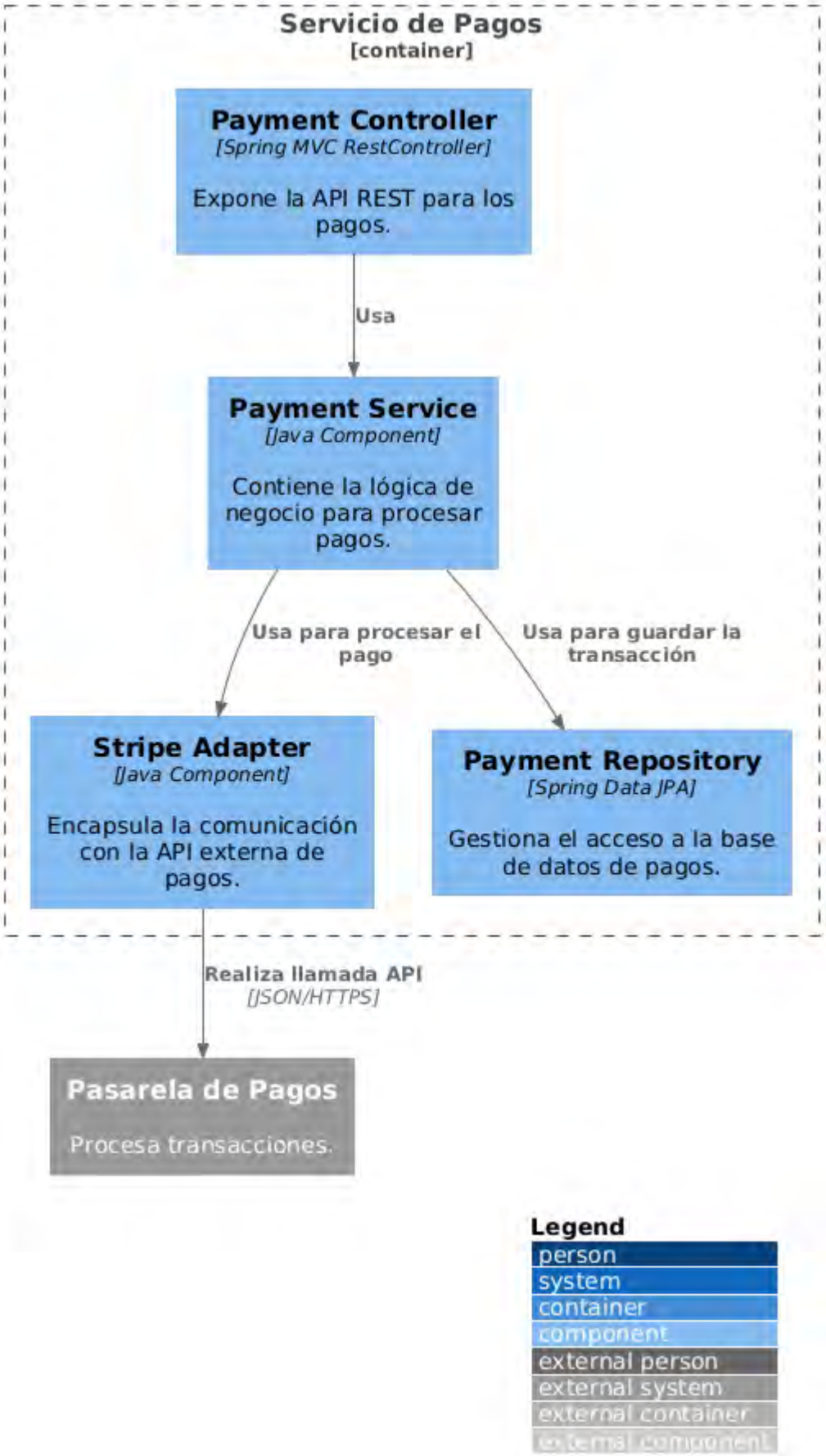
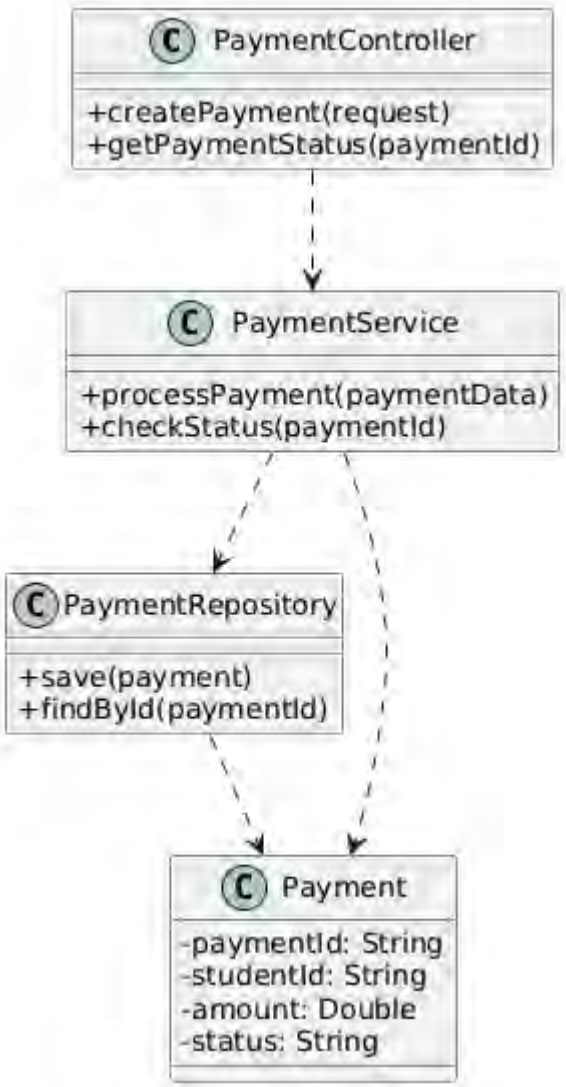
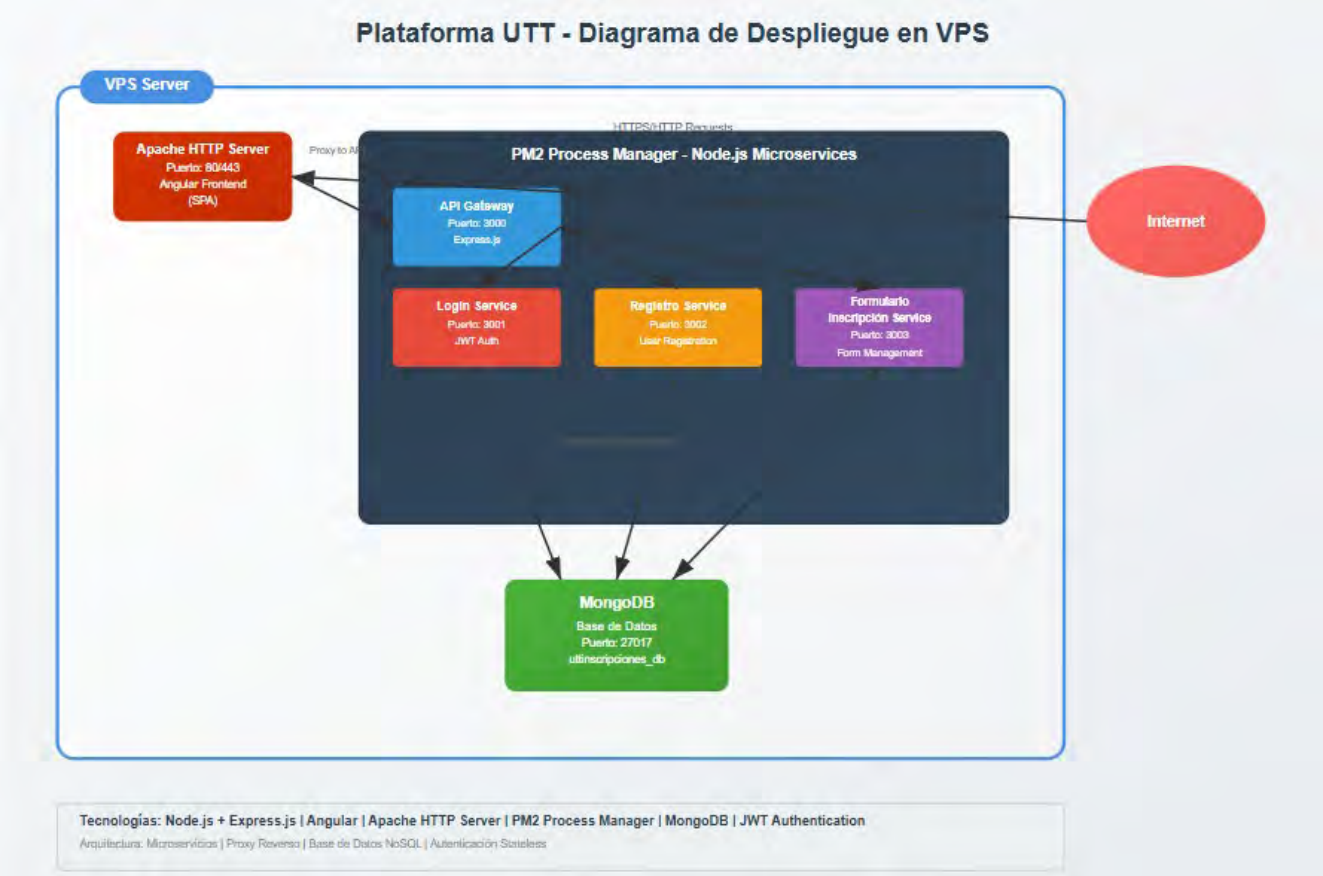

Servicio de Pagos
[container]
Payment Controller
[Spring MVC RestController]
Expone la API REST para los pagos.
Usa
Payment Service
[Java Component]
Contiene la lógica de negocio para procesar pagos.
Usa para procesar el pago
Usa para guardar la transacción
Stripe Adapter
[Java Component]
Encapsula la comunicación con la API externa de pagos.
Payment Repository
[Spring Data JPA]
Gestiona el acceso a la base de datos de pagos.
Realiza llamada API
[JSON/HTTPS]
Pasarela de Pagos
Procesa transacciones.
Legend
person
system
container
component
external person
external system
external container
external component

Diagrama de Código (Nivel 4)

El Diagrama de Código (representado aquí como un diagrama de clases UML) ofrece la vista más granular, mostrando la estructura estática del código para los componentes del Servicio de Pagos. Define las clases principales como `PaymentController` y `PaymentService`, sus métodos públicos y las relaciones de dependencia entre ellas. Este diagrama sirve como un plano para los desarrolladores que implementarán la funcionalidad del microservicio.





Decisiones arquitectónicas (ADR)
Documentar al menos **3 decisiones clave**

ADR 1: Adopción de una Arquitectura de Microservicios

Decisión: Se ha decidido dejar atrás el enfoque de una monolito y pasar a una arquitectura de microservicios. Esto significa que cada función principal del negocio (como la autenticación, las inscripciones y los pagos) funcionará como un servicio pequeño e independiente.

Contexto y Razones: El sistema actual tiene problemas importantes para crecer, mantenerse disponible y ser fácil de actualizar. En los momentos de mayor demanda, como en las inscripciones, todo el sistema se vuelve lento. Además, cualquier cambio pequeño nos obliga a actualizar toda la aplicación, un proceso que es lento y arriesgado. Esta decisión busca crear una base que nos permita crecer de manera ordenada.

- **Análisis a Corto Plazo:**
    - **Impacto:** Al principio, el desarrollo y la puesta en marcha serán más complejos. Necesitaremos configurar una infraestructura nueva, que incluye un API Gateway (una puerta de entrada única para todos los servicios), una herramienta para manejar los contenedores y un sistema para monitorear que todo funcione correctamente.
    - **Ventaja:** Permite que los equipos de desarrollo trabajen al mismo tiempo en diferentes servicios, lo que ayuda a entregar nuevas funciones más rápido desde el inicio.
- **Análisis a Largo Plazo:**
    - **Impacto:** Se necesitará una buena organización y cultura de DevOps para manejar la complejidad de tener muchos servicios funcionando a la vez.
    - **Ventaja:** Podremos escalar solo las partes que lo necesitan (por ejemplo, dar más capacidad al servicio de inscripciones en temporada alta). El mantenimiento se vuelve mucho más sencillo, ya que los equipos pueden actualizar sus servicios de forma independiente. También, el sistema se vuelve más resistente a fallos: si el servicio de pagos tiene un problema, los usuarios todavía podrán consultar los cursos.

ADR 2: Uso de una Base de Datos Compartida con Esquemas Lógicos Separados

Decisión: Cada microservicio gestionará su propio esquema (colecciones) dentro de una única instancia compartida de MongoDB. Aunque los servicios son independientes en su código y despliegue, accederán a la misma base de datos física. La comunicación directa entre servicios para obtener datos está prohibida; si un servicio necesita información de otro, deberá hacerlo a través de su API correspondiente.

Contexto y Razones: Para la etapa actual del proyecto, la complejidad de configurar, gestionar y mantener múltiples bases de datos (y posiblemente de diferentes tecnologías) supera los beneficios. Compartir una única base de datos simplifica enormemente la configuración inicial, el desarrollo y los costos de infraestructura, permitiendo un avance más rápido. Esta decisión ofrece un equilibrio entre la separación lógica de los servicios y la simplicidad operativa.

Análisis a Corto Plazo:

- **Impacto:** Se elimina la complejidad de la replicación de datos o el uso de patrones avanzados como Sagas para transacciones distribuidas. El desarrollo es más directo.
- **Ventaja:** La puesta en marcha del entorno de desarrollo es mucho más rápida. Todos los equipos trabajan con una única conexión a la base de

datos, lo que simplifica la configuración de las variables de entorno y la gestión de la infraestructura.

Análisis a Largo Plazo:

- **Impacto:** Existe el riesgo de que los servicios se acoplen a nivel de base de datos si no se respeta la disciplina de no acceder a colecciones ajenas. Una futura migración al patrón "Base de Datos por Servicio" requerirá un esfuerzo de refactorización considerable.
- **Ventaja:** Permite que el sistema evolucione rápidamente en sus primeras fases. Si en el futuro la escalabilidad lo exige, se puede planificar una migración progresiva de los servicios más críticos a sus propias bases de datos, ya que la arquitectura de microservicios ya está establecida a nivel de aplicación.

ADR 3: Uso de JSON Web Tokens (JWT) para la Autenticación

Decisión: Se usará un sistema de autenticación "sin estado" (que no guarda la sesión en el servidor) a través de JSON Web Tokens (JWT). Un servicio de autenticación creará estos tokens, y los clientes los enviarán en cada petición para que el API Gateway verifique su validez.

Contexto y Razones: En un sistema con muchos servicios, la forma tradicional de manejar las sesiones en el servidor se convierte en un obstáculo para poder crecer. Necesitamos un método que le permita a cualquier servicio saber quién es el usuario de forma segura y por sí mismo, sin depender de un lugar central donde se guarde la sesión.

- **Análisis a Corto Plazo:**
    - **Impacto:** Se necesita crear un proceso seguro para la creación, firma y validación de los tokens. Los desarrolladores deben conocer y aplicar las mejores prácticas de seguridad para evitar riesgos, como el manejo del tiempo de vida del token y su almacenamiento en el navegador.
    - **Ventaja:** Simplifica el diseño de los demás servicios, ya que estos ya no tienen la responsabilidad de gestionar las sesiones de los usuarios.
- **Análisis a Largo Plazo:**
    - **Impacto:** Invalidar un token antes de que expire (por ejemplo, cuando un usuario cierra sesión) es un reto que necesita soluciones extra, como una lista de tokens revocados.
    - **Ventaja:** La naturaleza "sin estado" de los JWT hace mucho más fácil que el sistema crezca (agregando más servidores). Se pueden añadir o quitar copias de cualquier servicio sin preocuparse por a qué servidor se conecta un usuario. Esto encaja perfectamente con una arquitectura moderna, pensada para la nube.