SISTEMAS DISTRIBUIDOS

SNEIDER MONROY QUIROGA

HUBER ANDRES PARRA

CAMILO ANDRES CHAVARRO

JUAN FERNANDO PEREZ

DOCENTE: JESUS ARIEL GONZALES BONILLA

Corporación universitaria Corhuila
Facultad de ingeniería
Programa de ingeniería de sistemas
Asignatura: sistemas distribuidos
Neiva, 4 de Noviembre 2025

Tabla de Contenido

I. DESCRIPCIÓN GENERAL

El proyecto University Academic Tracker se enmarca en el desarrollo de una solución tecnológica integral basada en sistemas distribuidos que permita a los estudiantes universitarios gestionar su rendimiento académico en tiempo real. La iniciativa combina los principios de arquitectura de microservicios y el enfoque de Aprendizaje Basado en Proyectos (ABP), con el objetivo de diseñar, implementar y evaluar una plataforma que optimice el seguimiento académico, la comunicación y la toma de decisiones dentro del ámbito universitario.

Este proyecto integra los conceptos teóricos vistos en la asignatura de Sistemas Distribuidos con la práctica profesional mediante el desarrollo colaborativo de un sistema real, fomentando habilidades técnicas, de trabajo en equipo y de autogestión del conocimiento. El desarrollo se lleva a cabo bajo una metodología Scrum combinada con Kanban, con sprints semanales, revisión de avances cada martes y entregas incrementales del producto funcional.

II. ROLES DEL EQUIPO

Integrante	***Rol***	***Responsabilidad Principal***
Juan Fernando Pérez Olaya	Product Owner (PO)	Definir requerimientos, priorizar backlog, validar entregas y gestionar el valor del producto.
Huber Andrés Parra Molina	Desarrollador frontend	Construcción de la interfaz web, diseño UX/UI, conexión con microservicios y despliegue visual.
Camilo Andrés Chavarro Guenis	Desarrollador backend	Implementación de microservicios, configuración del API Gateway y comunicación entre servicios.
Sneider Quiroga Monrroy	Desarrollador frontend	Gestión de base de datos, pruebas de servicios, control de errores y documentación técnica.

III. OBJETIVOS

Objetivo General

Desarrollar una plataforma académica distribuida que permita a los estudiantes universitarios gestionar, visualizar y analizar su rendimiento académico mediante microservicios escalables, aplicando metodologías ágiles para fomentar el aprendizaje práctico y colaborativo.

Objetivos específicos

- Diseñar la arquitectura distribuida del sistema basada en microservicios que permita escalabilidad y modularidad.

- Implementar microservicios dedicados a la gestión de estudiantes, universidades, docentes y calificaciones.

- Desarrollar un frontend web intuitivo que facilite la interacción y visualización de datos académicos.

- Integrar un servicio de autenticación seguro basado en JWT y OAuth2.

- Aplicar prácticas ágiles combinando Scrum y Kanban, con sprints semanales y revisión continua del progreso.

IV. ROLES DEL ESTUDIANTE

Investigador: Buscar, analizar y sintetizar información relevante para el proyecto, desarrollando habilidades de investigación y pensamiento crítico.

Colaborador: Participar activamente en el trabajo en equipo, aprendiendo a comunicarse, negociar y resolver problemas de manera colectiva.

Creador: Diseñar y elaborar soluciones tanto tangibles como intangibles, promoviendo la creatividad y la aplicación práctica del conocimiento adquirido.

Presentador: Comunicar los resultados y hallazgos del proyecto ante una audiencia, desarrollando habilidades de presentación y comunicación efectiva.

V. FASE 1. FORMULACIÓN DEL PROBLEMA

4.1 Análisis del contexto educativo universitario

En el entorno universitario moderno, gestionar eficientemente la información académica de estudiantes y docentes es fundamental para mejorar el rendimiento y la toma de decisiones. Los alumnos necesitan herramientas para monitorear sus calificaciones, establecer metas académicas y recibir retroalimentación oportuna sobre su progreso.

Frente a esta necesidad, el proyecto University Academic Tracker surge con el objetivo de proveer una plataforma académica integral que permita a los estudiantes gestionar sus calificaciones, porcentajes por corte, metas académicas y actividades de cada asignatura. El sistema está diseñado para ofrecer organización de la información académica, alertas preventivas sobre posibles riesgos académicos y un control en tiempo real del rendimiento académico.

En otras palabras, el contexto identificado es uno en el que la comunidad universitaria se beneficiará de un seguimiento más cercano y proactivo del desempeño académico, algo que las plataformas tradicionales a veces no logran proporcionar en tiempo real.

El análisis del problema reveló que muchos sistemas existentes carecen de funciones avanzadas como alertas tempranas de bajo rendimiento o visualización dinámica de estadísticas personalizadas. Por ejemplo, suele ser difícil para un estudiante saber si su promedio está por debajo del esperado antes de finalizar el semestre, o para un docente identificar rápidamente qué estudiantes necesitan apoyo adicional. Todo esto justifica la creación de una plataforma que centralice datos académicos y brinde funcionalidades interactivas de análisis y reporte. En síntesis, University Academic Tracker aborda problemas cotidianos en la gestión académica universitaria, proporcionando un espacio unificado donde estudiantes, docentes y administradores puedan interactuar con la información académica de manera ágil y efectiva, favoreciendo la toma de decisiones informadas y oportunas.

VI. Justificación del uso de sistemas distribuidos

Para satisfacer los requerimientos del contexto descrito, se decidió emplear una arquitectura de sistemas distribuidos basada en microservicios. Esta elección tecnológica se justifica por múltiples factores. En primer lugar, una arquitectura de microservicios ofrece escalabilidad, permitiendo que distintas partes de la aplicación crezcan de manera independiente según la demanda; por ejemplo, si el módulo de estadísticas requiere más recursos en época de exámenes, se puede escalar ese microservicio sin afectar a los demás. En segundo lugar, promueve la modularidad, ya que divide el sistema en componentes autónomos (servicios) alineados con diferentes dominios funcionales (estudiantes, docentes, autenticación, etc.), lo cual facilita significativamente el desarrollo y mantenimiento del software. Cada microservicio encapsula una funcionalidad específica de la plataforma (p. ej., gestión de estudiantes) de modo que las modificaciones o actualizaciones en un servicio no impactan directamente a los otros, contribuyendo a un mantenimiento sencillo y reduciendo el riesgo de efectos colaterales.

Además, en un entorno universitario con potencialmente miles de usuarios distribuidos (estudiantes de múltiples programas, docentes y administradores), una arquitectura distribuida resulta adecuada para asegurar la disponibilidad y tolerancia a fallos. Si un componente del sistema falla. los demás pueden continuar operando de forma independiente, evitando una caída total de la plataforma. Esta resiliencia está alineada con las mejores prácticas de sistemas distribuidos modernos, donde la fiabilidad se logra a través de la redundancia y la independencia de servicios. En suma, el uso de microservicios en University Academic Tracker no solo responde a requerimientos técnicos de escalabilidad horizontal y despliegue flexible, sino también a la necesidad de evolución a largo plazo: la arquitectura distribuida permite incorporar nuevos

módulos o funcionalidades (como servicios de analítica avanzada o integración con otras plataformas) sin tener que reestructurar un sistema monolítico entero.

VII. REQUISITOS FUNCIONALES Y NO FUNCIONALES

Derivado del análisis del contexto y la visión del proyecto, se establecieron tanto requisitos funcionales como no funcionales siguiendo la recomendación del estándar IEEE-830 para especificación de requerimientos de software. A continuación, se detallan los principales:

6.1 Requisitos Funcionales

6.1.1 RF1. Autenticación de usuarios: El sistema debe permitir que los usuarios se registren con una cuenta nueva, inicien sesión mediante credenciales (usuario y contraseña) y recuperen su contraseña a través de correo electrónico en caso de olvido. Durante el inicio de sesión se validarán campos obligatorios y se ofrecerá la opción de "recordar sesión" para mejorar la experiencia de usuario. Asimismo, se deberán mostrar mensajes de error claros cuando las credenciales sean inválidas o el usuario no exista, facilitando que el usuario comprenda la causa del fallo.

6.1.2 RF2. Control de acceso por roles: Tras autenticarse, el usuario accederá a un panel principal con menús y opciones adaptadas a su rol, ya sea estudiante, docente o administrador. El sistema mostrará únicamente las funcionalidades pertinentes a cada tipo de usuario (p. ej., un estudiante no verá opciones de administrar usuarios, y un administrador no verá opciones propias de estudiantes). Además, deberá existir la función de cierre de sesión seguro para proteger la cuenta del usuario.

6.1.3RF3. Gestión de estudiantes por parte del administrador: Los administradores de la plataforma podrán registrar nuevos estudiantes mediante formularios validados, consultar un listado de estudiantes (con capacidades de búsqueda y filtrado para encontrar información

rápidamente) y actualizar o editar los datos de un estudiante cuando sea necesario. También podrán eliminar estudiantes del sistema, con la precaución de requerir confirmación de la acción para evitar borrados accidentales. Cada estudiante registrado tendrá un perfil que incluye sus datos personales y académicos.

6.1.4 RF4. Gestión de docentes por parte del administrador: De forma análoga a los estudiantes, el administrador podrá registrar docentes en la plataforma a través de formularios validados, visualizar un listado de docentes con filtros de búsqueda, y efectuar la edición o eliminación de los registros de docentes, asegurando confirmación en las eliminaciones. Esto permite mantener actualizada la información del personal académico de la universidad.

6.1.5 RF5. Gestión de universidades y programas académicos: El sistema permitirá al administrador registrar nuevas universidades o instituciones académicas, así como sus programas o facultades asociadas. Al registrar una universidad se configurarán atributos académicos relevantes, tales como el número de cortes (períodos de calificación en el semestre) y la nota mínima aprobatoria, de forma que la institución quede correctamente parametrizada en el sistema. También se podrán consultar, editar y eliminar los datos de universidades y programas existentes, con funcionalidad de confirmación de eliminaciones para evitar errores.

6.1.6 RF6. Perfil del estudiante y consulta de calificaciones: Cada estudiante tendrá acceso a su perfil personal, donde podrá visualizar sus datos básicos y su información académica relevante (por ejemplo, carrera, semestre, etc.), lo cual le permite verificar que sus datos estén almacenados correctamente. Más importantemente, el estudiante podrá consultar sus notas obtenidas en cada asignatura y en cada actividad evaluativa de la misma. El sistema mostrará el desglose por cortes

(parciales) y calculará, cuando aplique, los promedios parciales y finales de cada asignatura. El estudiante también podrá consultar su promedio general acumulado en el semestre, e incluso comparar sus notas con los promedios grupales para entender su posición relativa dentro de la clase.

6.1.7 RF7. Seguimiento de rendimiento y progresión académica (estudiantes): La plataforma brindará a los estudiantes visualizaciones gráficas de su desempeño a lo largo del tiempo. Por ejemplo, el estudiante podrá ver gráficas de progreso académico por semestre para identificar tendencias de mejora o descenso en sus calificaciones. También dispondrá de una funcionalidad para simular la nota que necesita en futuras evaluaciones para alcanzar cierto objetivo (por ejemplo, cuál sería la nota mínima necesaria en el examen final para lograr aprobar la materia). Estas funcionalidades de análisis personal ayudan al alumno a planificar sus esfuerzos académicos.

6.1.8 RF8. Gestión de asignaturas, grupos y actividades: Los docentes podrán administrar las actividades evaluativas dentro de sus asignaturas. Esto incluye crear nuevas actividades (como exámenes, talleres, proyectos), asignarles un porcentaje de contribución a la nota final y definir fechas límite. Cada asignatura podrá tener varios grupos o secciones; el sistema permitirá manejar estos grupos para asociar estudiantes y docentes a clases específicas. Los estudiantes se inscribirán en grupos (p. ej., mediante la entidad de matrícula que relaciona a un estudiante con un grupo en un período determinado), lo que posibilitará gestionar las calificaciones de forma diferenciada por clase. Cuando un docente cree o edite una actividad, el sistema debe recalcular la ponderación total de la asignatura por corte, garantizando que los porcentajes asignados sumen 100%. De igual modo, existirá la opción de ingresar o actualizar las calificaciones de cada estudiante por actividad; esto podrá hacerse de forma manual por el docente o mediante cargas masivas de archivos (ver

RF9) para eficiencia.

6.1.9 RF9. Carga masiva de datos académicos desde archivos: Con el fin de agilizar la entrada de información, el sistema ofrecerá a los docentes la posibilidad de importar datos mediante archivos Excel o PDF. Específicamente, el docente podrá descargar plantillas predefinidas en Excel para el registro de estudiantes o de calificaciones, asegurando un formato estándar. Luego, podrá cargar al sistema un archivo Excel con la lista de estudiantes de un curso para registrarlos automáticamente en el sistema, o un archivo Excel con calificaciones para que el sistema procese y almacene esas notas asociándolas a cada estudiante y actividad correspondiente. Asimismo, el docente podrá subir archivos PDF con tablas de calificaciones exportadas de otros sistemas; en este caso, el sistema será capaz de interpretar y extraer los datos relevantes de las tablas en PDF, convirtiéndolos a un formato estructurado interno. En todos los casos, el sistema validará que el archivo subido cumpla con el formato esperado (por ejemplo, encabezados correctos, tipos de datos válidos). Si se detecta algún error o discrepancia en el archivo, se mostrará un mensaje de error claro indicando el problema, de modo que el docente pueda corregir el archivo y volverlo a intentar. Antes de aplicar definitivamente los cambios en la base de datos, el sistema mostrará una vista previa de los datos procesados para que el docente confirme que la información interpretada es correcta. Una vez confirmados, los datos extraídos se almacenarán en la base de datos de manera persistente para futuras consultas. Adicionalmente, el sistema mantendrá un historial de archivos cargados por el usuario, para posibilitar la trazabilidad de las importaciones realizadas y servir como evidencia ante posibles inconsistencias.

6.1.10 RF10. Estadísticas académicas: Una pieza central de la plataforma es la capacidad de analizar y reportar estadísticas del rendimiento académico. Los docentes podrán visualizar

estadísticas de notas por materia (promedios generales de cada grupo, comparación entre grupos, etc.), así como estadísticas por estudiante (rendimiento individual comparado con el resto). El sistema calculará medidas descriptivas como promedio, nota mínima, máxima y desviación estándar para un conjunto de calificaciones, permitiendo a los docentes identificar tendencias o anomalías en el desempeño de sus grupos. Estas estadísticas se podrán filtrar por diferentes criterios, por ejemplo por corte (periodo parcial) para analizar la evolución dentro de un mismo semestre o por rango de fechas. La plataforma presentará visualizaciones gráficas interactivas: gráficos de barras por materia (para ver comparaciones claras de promedios), gráficos circulares por grupo (para observar distribuciones de calificaciones), y permitirá alternar entre diferentes tipos de gráfico (líneas, barras, pastel) según la preferencia del usuario. Un caso de uso importante es que el docente pueda rápidamente identificar estudiantes en riesgo académico – por ejemplo, alumnos con promedio bajo o que hayan reprobado múltiples actividades – para tomar acciones preventivas. Para el nivel administrativo superior (coordinadores o administradores académicos), el sistema proporcionará métricas globales como el número total de estudiantes, docentes y universidades activas, con el fin de dar una visión general del estado de la plataforma educativa. También mostrarán gráficos de distribución globales (por ejemplo, distribución de estudiantes por programa o de calificaciones a nivel institucional) que ayuden a identificar patrones académicos a gran escala.

VIII. REQUISITOS NO FUNCIONALES

7.1.1 RNF1. Escalabilidad: El sistema debe ser escalable horizontalmente para soportar un creciente número de usuarios y datos sin degradar su desempeño. Gracias a la arquitectura de

microservicios adoptada, se puede lograr escalabilidad desplegando múltiples instancias de servicios críticos según la demanda. Esto es especialmente relevante en periodos de alta carga, como finales de semestre, donde el uso de funcionalidades de estadísticas puede aumentar considerablemente.

7.1.2 RNF2. Rendimiento y eficiencia: La plataforma debe mantener tiempos de respuesta adecuados, incluso al manejar volúmenes grandes de información (por ejemplo, consultar las calificaciones históricas de un estudiante o generar gráficos con miles de datos). Para lograrlo, se implementan mecanismos como la paginación en las vistas de datos extensos, de forma que la interfaz solo cargue subconjuntos manejables en cada consulta. Esto evita que la carga de información masiva afecte el rendimiento o bloquee la interacción del usuario. Adicionalmente, las operaciones de cálculo de estadísticas mejorando la eficiencia global del sistema.

7.1.3 RNF3. Seguridad y control de acceso: Dado que se maneja información académica sensible (calificaciones, datos personales de estudiantes), es crucial garantizar la seguridad. El sistema debe implementar autenticación robusta (con cifrado de contraseñas, tokens JWT/OAuth2 para sesiones, etc.) y autorización basada en roles para que cada usuario solo acceda a las funciones y datos que le corresponden. Las comunicaciones entre componentes deben realizarse preferentemente sobre canales seguros (HTTPS) y validarse todas las entradas de datos para prevenir vulnerabilidades comunes. Asimismo, se deberán registrar eventos de seguridad importantes (ej. intentos fallidos de autenticación) para auditoría.

7.1.4 RNF4. Usabilidad y diseño centrado en el usuario: La aplicación debe ofrecer una interfaz intuitiva y consistente, siguiendo lineamientos de UX/UI que faciliten su uso a los distintos

tipos de usuarios. Se empleará un framework visual (Vuetify con Vue.js) con componentes estandarizados para asegurar una experiencia uniforme. También se considerará la responsividad de la interfaz para que pueda ser accesible desde distintos dispositivos (computadores de escritorio, tabletas, móviles). Un objetivo no funcional clave es que los usuarios finales (alumnos, profesores, admins) adopten fácilmente la herramienta sin requerir una extensa capacitación.

7.1.5 RNF5. Mantenibilidad y extensibilidad: Gracias a la modularidad de la arquitectura, el sistema debe ser fácil de mantener y extender. Cada microservicio cuenta con un código base delimitado a su contexto, lo que facilita localizar y corregir errores o realizar mejoras sin afectar a otros módulos. Se adoptaron estándares de codificación y buenas prácticas desde el inicio (por ejemplo, formateo de código consistente, linters, etc., en el frontend y backend). Además, se integró una herramienta de análisis de calidad de código (SonarQube) en los proyectos backend de Student, Teacher y University, para identificar automáticamente bugs, code smells o vulnerabilidades, y asegurar que todos los módulos mantengan estándares de calidad homogéneos. Esto contribuye a la sostenibilidad del proyecto a largo plazo, ya que un código de alta calidad es más fácil de entender, probar y modificar. Igualmente, el diseño orientado a microservicios permite añadir nuevas funcionalidades (por ejemplo, un futuro módulo de analítica predictiva o un servicio móvil) incorporando nuevos servicios o ampliando los existentes con un impacto controlado.

7.1.6 RNF6. Fiabilidad y tolerancia a fallos: El sistema debe ser confiable, minimizando el tiempo fuera de servicio. En producción, se contemplará la implementación de instancias redundantes de servicios críticos y un servidor de descubrimiento (Eureka) para gestionar la disponibilidad de los microservicios. Si un microservicio específico dejara de funcionar, el sistema

global debería continuar operando parcialmente y, en la medida de lo posible, recuperarse automáticamente (por ejemplo, reintentar conexiones o redistribuir la carga). El uso de un API Gateway central permitirá también manejar de forma uniforme los errores y las caídas de servicios backend, retornando respuestas controladas al cliente en caso de incidencias en lugar de que la aplicación falle de forma inesperada.

En conjunto, estos requisitos funcionales y no funcionales delinean un sistema robusto, centrado en las necesidades del usuario universitario y construido sobre bases técnicas sólidas. A continuación, se detalla el análisis y diseño de la solución propuesta que cumple con dichos requerimientos.

IX. FASE 2. ANÁLISIS Y DISEÑO DE LA SOLUCIÓN

Tras identificar claramente el problema y sus requisitos, se procedió a la fase de análisis y diseño, donde se definió la estructura técnica del University Academic Tracker. Durante esta fase se optó por una arquitectura distribuida de microservicios, se seleccionaron las tecnologías adecuadas para cada componente y se planificó el trabajo en sprints usando Scrum. Asimismo, se produjeron diversos diagramas de diseño (casos de uso, clases, componentes y secuencia).

Para documentar la solución propuesta. A continuación, se explica la arquitectura general y se integran los diagramas mencionados junto con la justificación de las principales decisiones de diseño.

X. EXPLICACIÓN DE LA ARQUITECTURA DISTRIBUIDA

La arquitectura distribuida del sistema sigue el modelo de microservicios independientes que cooperan entre sí. Cada microservicio corresponde a un módulo funcional del dominio universitario, aislando las responsabilidades en unidades desplegables por separado. En particular, se definieron los siguientes servicios principales (cada uno con su propio repositorio de código):

9.1.1 Auth Service (Servicio de Autenticación): responsable de la autenticación y autorización de usuarios, manejando las credenciales (usuarios/contraseñas) y la emisión de tokens de acceso (JWT/OAuth2). Este servicio centraliza el inicio de sesión y registro, y valida los roles de usuario al momento de ingresar al sistema.

9.1.2 Student Service (Servicio de Estudiantes): gestiona la información de estudiantes y sus calificaciones. Provee APIs para registrar/actualizar datos de estudiantes y para consultar las notas y promedios de cada estudiante en sus cursos. También implementa la lógica de cálculo de indicadores académicos individuales (por ejemplo, cálculo de promedio general del estudiante).

9.1.3 Teacher Service (Servicio de Docentes): se encarga de la gestión de los docentes y las asignaturas que imparten. Expone funcionalidades para CRUD de docentes y potencialmente para manejar las actividades evaluativas y calificaciones desde la perspectiva del profesor (por ejemplo, ingresar notas de un grupo o consultar estadísticas de su curso).

9.1.4 University Service (Servicio de Universidades): administra los datos de las universidades (o instituciones) y sus programas académicos. Incluye operaciones para CRUD de universidades, facultades/programas, y posiblemente las materias ofrecidas en cada programa. Este servicio sienta las bases para que el sistema soporte múltiples instituciones y estructuras curriculares diferentes.

9.1.7API Gateway: un gateway central que funge como punto de entrada único para todos los clientes de la plataforma. El API Gateway enruta las solicitudes hacia el microservicio correspondiente (por ejemplo, peticiones relacionadas con estudiantes van al Student Service) y puede aplicar políticas transversales de seguridad, rate limiting, etc.. Esto simplifica la comunicación cliente-servidor ya que el frontend solo necesita conocer la URL del gateway y este se encarga de distribuir el tráfico internamente.

9.1.8 Eureka Server (Service Discovery): un servidor de registro y descubrimiento de servicios utilizado para que todos los microservicios se registren con su dirección. De esta manera, el API Gateway y los demás servicios pueden descubrir dinámicamente las ubicaciones de los microservicios a través de Eureka. Esto facilita la elasticidad del sistema: si un microservicio escala a múltiples instancias, todas ellas se registran en Eureka y el gateway puede balancear la carga entre ellas.

9.1.9 Config Server (Servidor de Configuración Centralizada): previsto para centralizar la configuración de todos los microservicios (por ejemplo, cadenas de conexión a bases de datos, puertos, credenciales de servicios externos, etc.). Un Config Server permite mantener la consistencia de configuraciones a través de los entornos y actualizar parámetros sin necesidad de redeployar cada servicio individualmente. Este componente también estaba planificado como parte

de la arquitectura final pero quedó pendiente de implementación al culminar el desarrollo inicial.

9.1.10 Frontend Web: aunque no es un microservicio de backend, merece mención la aplicación frontend desarrollada en Vue.js (con el framework de componentes Vuetify) que consume las APIs del backend. Esta aplicación se despliega de manera independiente y se comunica con el API Gateway para obtener los datos y enviar acciones de usuario. Al ser una Single Page Application, maneja rutas internas (páginas) para las diferentes vistas de la plataforma.

La interacción entre estos componentes está diseñada de forma que se minimicen las dependencias acopladas. En general, se sigue un patrón de comunicación REST síncrona a través del API Gateway: el frontend realiza peticiones HTTP al Gateway, este las redirige al microservicio correspondiente, y la respuesta fluye de regreso por el mismo camino. Internamente, algunos microservicios también se comunican entre sí. Por ejemplo, cuando el Student Service necesita datos de la estructura académica (como la lista de materias de un programa), puede realizar una petición al University Service; o cuando el Teacher Service registra calificaciones de varios estudiantes, podría notificar al Student Service para que recalcule promedios globales. Estas interacciones se orquestan a través de APIs bien definidas para cada servicio, documentadas y versionadas en los repositorios respectivos.

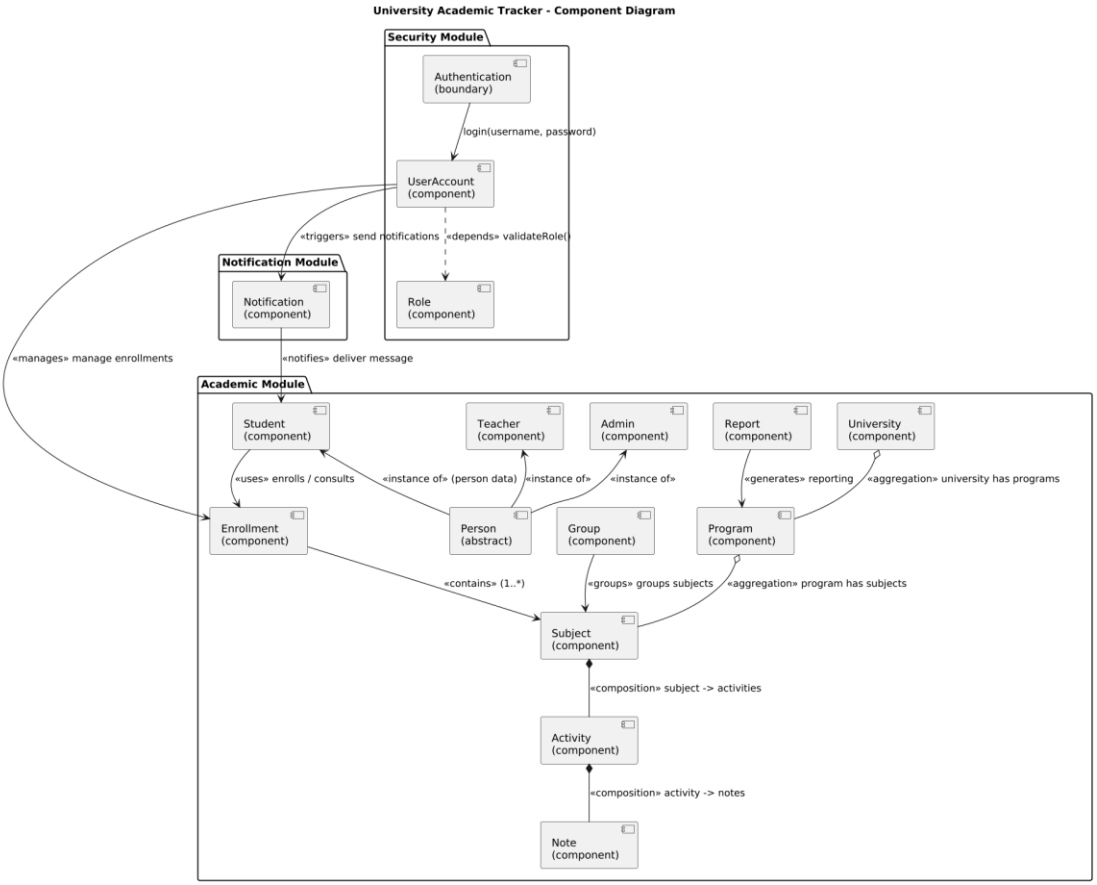


Figura 1. Diagrama de componentes (arquitectura distribuida) del University Academic Tracker.

En la Figura 1 se puede apreciar cómo el módulo Académico agrupa la lógica central del sistema (gestión de estudiantes, docentes, universidades, grupos, matrículas, actividades y notas), mientras que el módulo de Seguridad (Auth Service) .Todos estos servicios están concebidos para desplegarse separadamente y escalar según sus necesidades de carga. La separación de preocupaciones es evidente: el módulo de Seguridad se encarga de validar credenciales y mantener sesiones seguras, liberando al módulo Académico de esa responsabilidad

Desde una perspectiva de despliegue, cada microservicio puede residir en un contenedor o servidor distinto, comunicándose a través de la red (generalmente protocolos HTTP/HTTPS para las APIs REST). La presencia del Service Discovery (Eureka) y del API Gateway añade robustez: los servicios reportan su estado y dirección a Eureka, de modo que si alguno cambia (por ejemplo, una nueva instancia de Student Service aparece en un puerto diferente), el Gateway lo sabrá y podrá encaminar tráfico hacia ella. De igual manera, si algún servicio se cae o deja de responder, Eureka lo marcará como no disponible y el Gateway detendrá las solicitudes hacia ese destino, incrementando la tolerancia a fallos del sistema en su conjunto.

XI. JUSTIFICACIÓN DE LA SELECCIÓN TECNOLÓGICA

La selección de tecnologías para implementar la solución se realizó considerando la experiencia del equipo, la idoneidad de cada herramienta para los requerimientos y la integración entre componentes. A nivel de backend, se optó por Java con Spring Boot como marco de desarrollo. Spring Boot es ampliamente utilizado en la construcción de microservicios por su soporte integrado para crear APIs RESTful, su manejo de dependencias (Spring Starter) y facilidades para desplegar servicios independientes rápidamente. Además, Spring Boot ofrece módulos complementarios (Spring Security, Spring Data, Spring Cloud) que encajan con las necesidades del proyecto: por ejemplo, Spring Security para implementar JWT en el Auth Service, Spring Cloud Netflix/Eureka para el descubrimiento de servicios, y Spring Data JPA para interaccionar con la base de datos de manera sencilla. La decisión de crear proyectos separados para cada microservicio en Java/Spring Boot fue documentada desde el Sprint 3, donde se configuró el proyecto base backend y se definió la base de datos inicial con las entidades fundamentales. Cada microservicio tendría su propio esquema de base de datos, con la posibilidad

de usar un motor relacional (p. ej. PostgreSQL o MySQL) para garantizar la consistencia de los datos académicos. El uso de Java, un lenguaje tipado y con amplio soporte empresarial, brinda robustez y rendimiento, a la vez que facilita encontrar recursos (librerías, documentación) para casos de uso comunes en aplicaciones universitarias.

En el frontend, la tecnología seleccionada fue Vue.js, un framework progresivo de JavaScript, complementado con Vuetify (un conjunto de componentes UI basados en Material Design). Esta elección responde a la necesidad de construir una interfaz de usuario dinámica y responsiva, que ofrezca una experiencia de una sola página (SPA) fluida. Vue.js es conocido por su curva de aprendizaje amigable y su fácil integración con otras librerías, lo cual permitió al equipo iniciar rápidamente el desarrollo del front. De hecho, en el Sprint 3 se creó el proyecto base en Vue con Vuetify y se configuraron las buenas prácticas de desarrollo front-end (linters como ESLint, formateador Prettier, y una estructura de carpetas organizada). Vuetify aportó componentes pre-diseñados (tablas, formularios, diálogos, gráficos básicos, etc.) que aceleraron la maquetación de las vistas siguiendo una estética moderna y uniforme. Esto fue importante para cumplir con el requisito de usabilidad: en lugar de diseñar cada elemento desde cero, se aprovecharon componentes probados que aseguran consistencia visual y compatibilidad entre navegadores.

En cuanto a las bases de datos, aunque no se especifica explícitamente en los documentos, se deduce la utilización de una base de datos relacional para las entidades principales (estudiantes, docentes, notas, etc.), dado que se mencionan almacenar datos en una base de datos relacional y la naturaleza relacional de las entidades (estudiante-inscrito en-grupo, grupo-pertenece a-materia, etc.). Es probable que se haya utilizado PostgreSQL o MySQL/MariaDB, comunes en entornos académicos, junto con JPA/Hibernate para el mapeo objeto-relacional en los microservicios Java.

Esto permite garantizar integridad referencial y realizar consultas SQL eficientes para estadísticas (por ejemplo, calcular promedios agrupados por grupo o materia).

Otras herramientas seleccionadas incluyen el uso de Git y GitHub para el control de versiones, con un repositorio central creado desde el inicio del proyecto para gestionar el código fuente y facilitar la colaboración entre los integrantes. GitHub además posibilitó la integración con pipelines de CI en caso de necesitarlas, e integraciones con SonarQube para análisis estático. Precisamente, la incorporación de SonarQube se hizo para los principales módulos backend (Student, Teacher, University) como parte de las mejoras continuas en calidad, lo que refleja la importancia dada a la mantenibilidad (RNF5). En ambiente de desarrollo, se usó Postman como cliente de pruebas para verificar las APIs de cada microservicio desplegado localmente, conforme lo planificado en historias de usuario de despliegue inicial. Esto permitió probar cada servicio de forma aislada antes de integrarlo con el frontend.

En resumen, las tecnologías seleccionadas (Java/Spring Boot para microservicios, Vue/Vuetify para la interfaz, base de datos relacional, herramientas de calidad y colaboración) fueron las más adecuadas para cumplir con los objetivos del sistema: ofrecen un balance entre robustez, facilidad de desarrollo y futuro crecimiento. Esta combinación tecnológica es común en aplicaciones empresariales modernas, garantizando que el University Academic Tracker se construya sobre una plataforma confiable y escalable.

XII. PLANEACIÓN SCRUM Y DISTRIBUCIÓN DE ROLES

El proyecto se gestionó utilizando la metodología Scrum, lo que implicó iteraciones cortas de una semana (sprints) y una adaptación continua a los cambios. Al inicio del desarrollo, el Product Owner definió las épicas e historias de usuario basándose en los requisitos identificados, estructurando así el product backlog. La planificación inicial estableció varias épicas alineadas con los cortes académicos y componentes del sistema, y se procedió a asignar historias de usuario a sprints concretos de acuerdo a las prioridades y dependencias. Las épicas utilizadas fueron las siguientes:

11.1.1 Épica 1: Definición y preparación del proyecto – Enfocada en sentar las bases (visión, alcance, arquitectura general, creación de repositorios, primeros diagramas y prototipos).

11.1.2 Épica 2: Base del Sistema – Enfocada en funcionalidades núcleo del sistema, principalmente la autenticación y la estructura básica del frontend/backend.

11.1.3 Épica 3: Gestión de Entidades – Enfocada en implementar las operaciones CRUD para las entidades principales (estudiantes, docentes, universidades) y asegurar su visualización e integración en la plataforma.

11.1.4 Épica 4: Análisis y Estadísticas – Centrada en desarrollar las funcionalidades de cálculo de estadísticas, generación de gráficos y visualización de datos de rendimiento académico.

11.1.5 Épica 5: Carga de Archivos – Orientada a habilitar la carga masiva de datos (estudiantes y notas) mediante archivos Excel/PDF y su procesamiento automatizado.

11.1.6 Épica 6: Módulo AuthService – Desarrollo e integración del servicio de autenticación central (login/registro con JWT).

11.1.7 Épica 7: Módulo API Gateway – Implementación del gateway para unificar la entrada al sistema y manejo de rutas a microservicios.

11.1.8 Épica 8: Módulo Eureka – Puesta en marcha del servidor de descubrimiento de servicios para orquestar la comunicación entre microservicios.

Cada épica se desglosó en historias de usuario (HU) más manejables. Por ejemplo, la Épica 3 (Gestión de Entidades) comprende historias como "registrar estudiantes (HU-205)", "editar estudiante (HU-207)", "registrar docentes (HU-214)", etc., mientras que la Épica 4 (Análisis y Estadísticas) abarca historias como "consultar promedio general (HU-301)" o "ver gráficos de distribución de notas (HU-306)". Estas historias de usuario fueron documentadas en el repositorio (archivo user-stories.txt) y también gestionadas en una herramienta Jira para seguimiento. El uso de Jira permitió al equipo organizar las historias en cada sprint, visualizar su estado (Pendiente, En Progreso, Hecho) y mantener la trazabilidad entre la planificación y la implementación real.

En cuanto a la distribución de roles dentro del equipo Scrum, se contaba con al menos las siguientes figuras (algunas personas asumieron múltiples roles dada la naturaleza académica del proyecto):

11.2.1 Product Owner: Encargado de establecer la visión del producto, priorizar el backlog y validar que lo desarrollado cumpliera con las expectativas. En las historias de usuario se ve reflejado este rol definiendo requisitos (por ejemplo, HU-002 donde el Product Owner organiza épicas e historias).

11.2.2 Scrum Master: Responsable de facilitar las ceremonias Scrum (reuniones de planificación, dailies, retrospectivas) y remover impedimentos. Aunque no se menciona explícitamente en los documentos, es común que un miembro del equipo haya desempeñado esta función para asegurar el cumplimiento de la metodología.

11.2.3Equipo de Desarrollo: Un grupo multifuncional que abarcó tanto desarrollo frontend como backend, así como tareas de documentación y pruebas. Dentro del equipo, se identificaron roles particulares según las habilidades:

11.2.4Desarrollador Backend: Enfocado en implementar los microservicios en Spring Boot, definir las entidades JPA, servicios y controladores REST. Por ejemplo, HU-007 describe la configuración del proyecto Java/Spring Boot, y luego historias como HU-202, HU-211, HU-219 relatan la configuración de los módulos Student, Teacher y University respectivamente. Igualmente, HU-203, HU-212, HU-220 cubren el desarrollo de las funciones CRUD en esos módulos.

11.2.5Desarrollador Frontend: Encargado de la implementación de la aplicación Vue.js, creación de componentes de interfaz y consumo de APIs. HU-006 documenta la creación del proyecto base Vue con sus buenas prácticas. Luego, historias como HU-101 (configuración de router y rutas seguras), HU-106 (vista de panel según rol) o HU-C02 (navegación entre vistas frontend), muestran actividades típicas del desarrollador frontend para estructurar la aplicación.

La planificación en cada sprint se realizó seleccionando un conjunto de historias de usuario del backlog que el equipo se comprometía a completar en la semana. Gracias a la granularidad de las historias, se conseguía entregar incrementos funcionales al final de cada sprint, por pequeños

que fueran. Por ejemplo, para el Sprint 1 se definieron objetivos de entendimiento y preparación (HU-001, HU-002); en Sprint 2 se abordó el diseño arquitectónico y documentación (HU-003, HU-004); en Sprint 3 se iniciaron las bases de la implementación técnica (proyectos base frontend y backend, HU-006, HU-007, HU-008); hacia Sprint 5 se cubrieron simultáneamente partes de la Base del Sistema (autenticación) y la Gestión de Entidades (inicio de módulos backend CRUD), y así sucesivamente. En algunos casos, se introdujeron historias de corrección o mantenimiento (HU-C##) fuera de la línea principal de cada épica, para solventar deudas técnicas o integrar herramientas (por ejemplo, HU-C01 traducir diagramas al inglés, HU-C03 conectar frontend con servicios, HU-C04/C05/C06 agregar paginación, HU-C07–C09 integrar SonarQube). Estas historias "C" evidencian la flexibilidad de Scrum para ajustar el backlog según necesidades emergentes y asegurar que al final del proyecto no solo se entreguen funciones, sino también un producto de calidad.

En términos de colaboración, al ser Scrum un marco ágil, se fomentó la comunicación constante dentro del equipo. Hubo reuniones diarias de seguimiento (daily stand-ups) para sincronizar el trabajo y resolver impedimentos rápidamente. La herramienta de Jira brindó transparencia, de modo que todos podían ver qué estaba haciendo cada miembro, y el Product Owner podía seguir el progreso y re-priorizar si era necesario. Al cierre de cada sprint, se hacían demostraciones de lo desarrollado (review) y reflexiones sobre cómo mejorar el proceso (retrospectiva), lo cual llevó a implementar mejoras continuas en la forma de trabajo del equipo.

En síntesis, la planeación Scrum permitió entregar el proyecto en incrementos manejables, con un control claro de qué funcionalidad entraba en cada sprint y quién era responsable de cada tarea. La distribución de roles aseguró que se cubrieran todas las áreas (desde la concepción del

producto hasta la implementación de código y pruebas), y que cada integrante aportara desde su especialidad para cumplir con la visión global. Este enfoque colaborativo y estructurado fue clave para el éxito del desarrollo de University Academic Tracker.

XIII. Diagrama de Casos de Uso

Como parte del análisis de requisitos, se elaboró un diagrama de casos de uso que resume las interacciones típicas de los distintos actores con el sistema. Este diagrama ayuda a visualizar de forma general qué puede hacer cada tipo de usuario (estudiante, docente, administrador/coordinador) en la plataforma y qué funcionalidades ofrece el sistema para cumplir con sus objetivos.

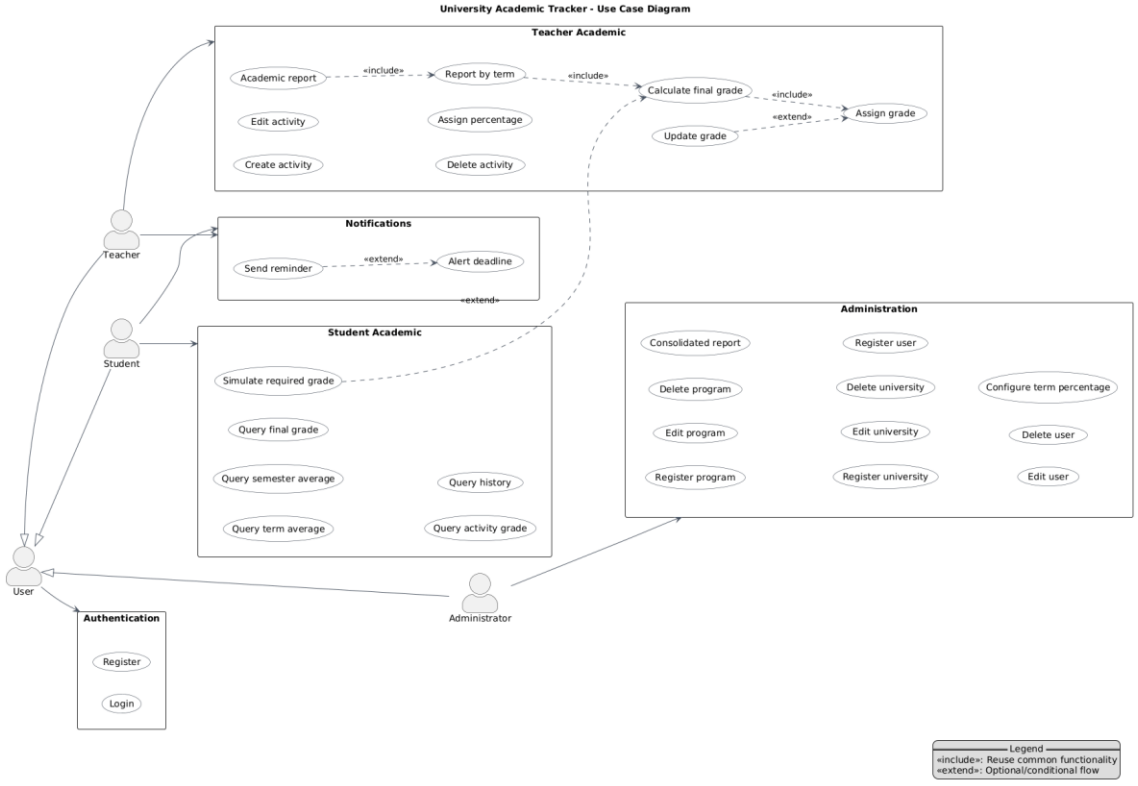


Figura 2. Diagrama de casos de uso del University Academic Tracker.

El diagrama de casos de uso (Figura 2) refleja de manera concisa las funcionalidades

delineadas en los requisitos funcionales. Vemos que:

13.1 FIGURA

12.1.1 El Estudiante tiene casos de uso orientados a la consulta y auto-gestión: puede ver su promedio académico general, consultar sus notas por materia y compararlas con el promedio del grupo, revisar un historial de calificaciones (por ejemplo de semestres anteriores) y usar una función de simulación para calcular qué nota necesitaría en futuras evaluaciones para alcanzar cierto promedio. Esto último es particularmente útil para que el estudiante planifique su estudio una vez que conoce sus calificaciones actuales.

12.1.2 El Docente cuenta con casos de uso tanto de gestión como de análisis: puede asignar calificaciones a los estudiantes (y actualizarlas si es necesario), administrar las actividades evaluativas de su curso (crear/editar/eliminar actividades y asignar sus porcentajes), y una vez ingresadas las notas, calcular automáticamente las notas finales de cada estudiante en la materia. En la parte analítica, el docente puede consultar estadísticas del rendimiento: promedios por materia, desempeño individual de estudiantes, distribución de notas (lo que puede ayudarle a ver la dispersión o detectar si muchos estudiantes están bajos en cierta actividad) .

12.1.3 El Administrador/Coordinador se encarga de la administración global: registrar y gestionar usuarios (tanto estudiantes como docentes, lo cual incluye dar de alta nuevos ingresos, actualizar información o dar de baja egresados), administrar programas académicos (crear o modificar programas, asignarles coordinadores, etc.), y administrar universidades en caso de que la plataforma maneje múltiples instituciones. Además, tiene casos de uso de configuración académica, como por ejemplo ajustar los porcentajes de cada corte evaluativo según la política de

la universidad (p. ej., 3 cortes de 30%, 30%, 40% respectivamente, o introducir un cuarto corte extraordinario). En el frente analítico, el administrador puede generar un reporte consolidado con datos de una facultad o de toda la universidad, y consultar métricas globales del sistema (cantidad de usuarios activos, comparativas entre distintas carreras, etc.), funciones que apoyan la toma de decisiones a nivel institucional.

Finalmente, todos los actores comparten los casos de uso de Autenticación: Registrarse en el sistema (solo aplicable para nuevos usuarios, típicamente estudiantes que crean cuenta, o posiblemente auto-registro de docentes) e Iniciar Sesión. Estos casos de uso son fundamentales, ya que controlan el acceso a cualquier otra función del sistema. El caso Registrarse incluiría validación de datos y asignación de un rol (por defecto, podría asignar rol "Estudiante" al registrarse, mientras que los roles de Docente o Administrador seguramente se gestionen internamente por el personal autorizado). Iniciar sesión simplemente verifica credenciales y redirige al panel adecuado según el rol.

En conclusión, el diagrama de casos de uso sirvió como guía para asegurarse de que todas las funcionalidades requeridas estuvieran identificadas y asignadas a un tipo de usuario claro. Cada caso de uso luego fue refinado en historias de usuario detalladas (como vimos en requisitos funcionales), asegurando la trazabilidad desde esta visión general hasta la implementación concreta en el proyecto.

XIV. DIAGRAMA DE CLASES

Para avanzar del análisis a un diseño de solución más concreto, se construyó un diagrama de clases que modela las entidades principales del sistema y sus relaciones. Este diagrama pertenece al diseño orientado a objetos de la aplicación, especialmente del dominio del problema, y sirvió como base para la creación de las entidades en la base de datos y los objetos de negocio en los microservicios. A continuación, se describe el modelo de clases propuesto y luego se presenta el diagrama correspondiente.

Las clases principales identificadas corresponden directamente a los conceptos clave del dominio académico:

Person (Persona)

clase general que encapsula atributos comunes a cualquier persona en el sistema (por ejemplo, nombre, identificación, correo electrónico). De esta clase derivan:

13.1.1 Student (Estudiante): extiende de Persona e incorpora atributos específicos del estudiante (por ejemplo, código de estudiante, programa académico al que pertenece, etc.).

13.1.2Teacher (Docente): extiende de Persona con atributos propios del profesor (por ejemplo, departamento o especialidad).

13.1.3 Admin (Administrador): podría modelarse también extendiendo Persona, representando a usuarios administrativos con privilegios especiales.

14.1 UserAccount

(CuentaUsuario): representa las credenciales de acceso al sistema, con atributos como username, email y password (hashed). Está asociada a Person de modo que cada persona del sistema tenga una cuenta de usuario para autenticarse. La relación es de uno a uno: un Student, Teacher o Admin tendrá exactamente un UserAccount, y esta cuenta está ligada a su persona.

14.1.2 Role (Rol): clase que define los tipos de rol que un UserAccount puede tener (estudiante, docente, admin, coordinador, etc.). Aunque en la implementación podría no ser necesaria una entidad de rol separada (bastando con un campo rol en UserAccount), en el diseño se consideró para permitir cuentas con roles múltiples en caso requerido. Un UserAccount puede tener uno o varios Roles asignados (relación muchos a muchos típicamente), lo que se denota en el diagrama con asociaciones entre UserAccount y Role.

14.1.3 University (Universidad): entidad que representa una institución educativa en el sistema. Tiene atributos como nombre, y parámetros académicos (p. ej., número de cortes por semestre, nota mínima aprobatoria, etc., que se mencionan en HU-222). Una University se relaciona con uno o varios Programas académicos.

14.1.4 Program (Programa): representa un programa académico (carrera o plan de estudios) ofrecido por una universidad. Atributos incluyen nombre del programa, facultad o departamento al que pertenece, etc. La relación entre University y Program es de uno a muchos: una universidad contiene múltiples programas.

14.1.5 Subject (Materia/Asignatura): clase que modela una asignatura específica que se imparte dentro de un programa. Tiene atributos como nombre de la materia, número de créditos, etc. Un Program típicamente ofrece muchas Materias. En el diagrama, Program está asociado con Subject indicando que un programa académico incluye varias materias (posiblemente una relación de agregación).

14.1.6 Group (Grupo): representa un grupo o sección particular en la que se imparte una materia durante un periodo (por ejemplo, "Matemáticas 101 - Grupo 1, Semestre 2025-1"). Atributos podrían ser identificador del grupo, periodo (semestre), horario, etc. La relación entre Subject y Group suele ser de uno a muchos: una materia puede tener varios grupos paralelos en un semestre. Cada Group estará asociado a un Teacher (el docente que imparte esa sección) y contendrá varios estudiantes inscritos.

14.1.7 Enrollment (Matrícula/Inscripción): clase que sirve de unión (asociativa) entre Student y Group, indicando que un estudiante está inscrito en un determinado grupo de una materia. Enrollment puede tener atributos como fecha de inscripción, estado (activa, retirada), y es fundamental para saber qué estudiantes pertenecen a qué clase. La cardinalidad refleja que un Student puede tener muchas Enrollment (una por cada grupo/materia que curse), y un Group tiene muchas Enrollment (una por cada estudiante inscrito). Esta entidad permite mantener un histórico de inscripciones por semestre.

14.1.8 Activity (Actividad Evaluativa): representa una tarea, examen, proyecto u otro elemento calificable dentro de una materia (y específicamente dentro de un grupo). Tiene atributos como título de la actividad, tipo, porcentaje que aporta al corte o al total de la materia, fecha de entrega, etc. Un Group tendrá asociadas múltiples Activities a lo largo del periodo (por ejemplo: Parcial 1, Taller 1, Proyecto final, etc.), generalmente organizadas por corte. Esta relación se puede modelar como un Group que contiene muchas Activities.

14.1.9 Note (Calificación): representa la calificación obtenida por un estudiante en una Activity específica. Tiene atributos como valor de la nota (número), posiblemente observaciones o comentarios del profesor. La clase Note se asocia tanto con Activity como con Student (o con Enrollment) para vincular quién obtuvo esa nota. En un modelo normalizado, Note podría tener una relación muchos a uno con Activity (una actividad tiene muchas notas, una por estudiante) y muchos a uno con Enrollment (una matrícula de estudiante en el grupo tiene muchas notas, una por cada actividad). Esto permite saber que "el estudiante X en el grupo Y obtuvo Z nota en la actividad A".

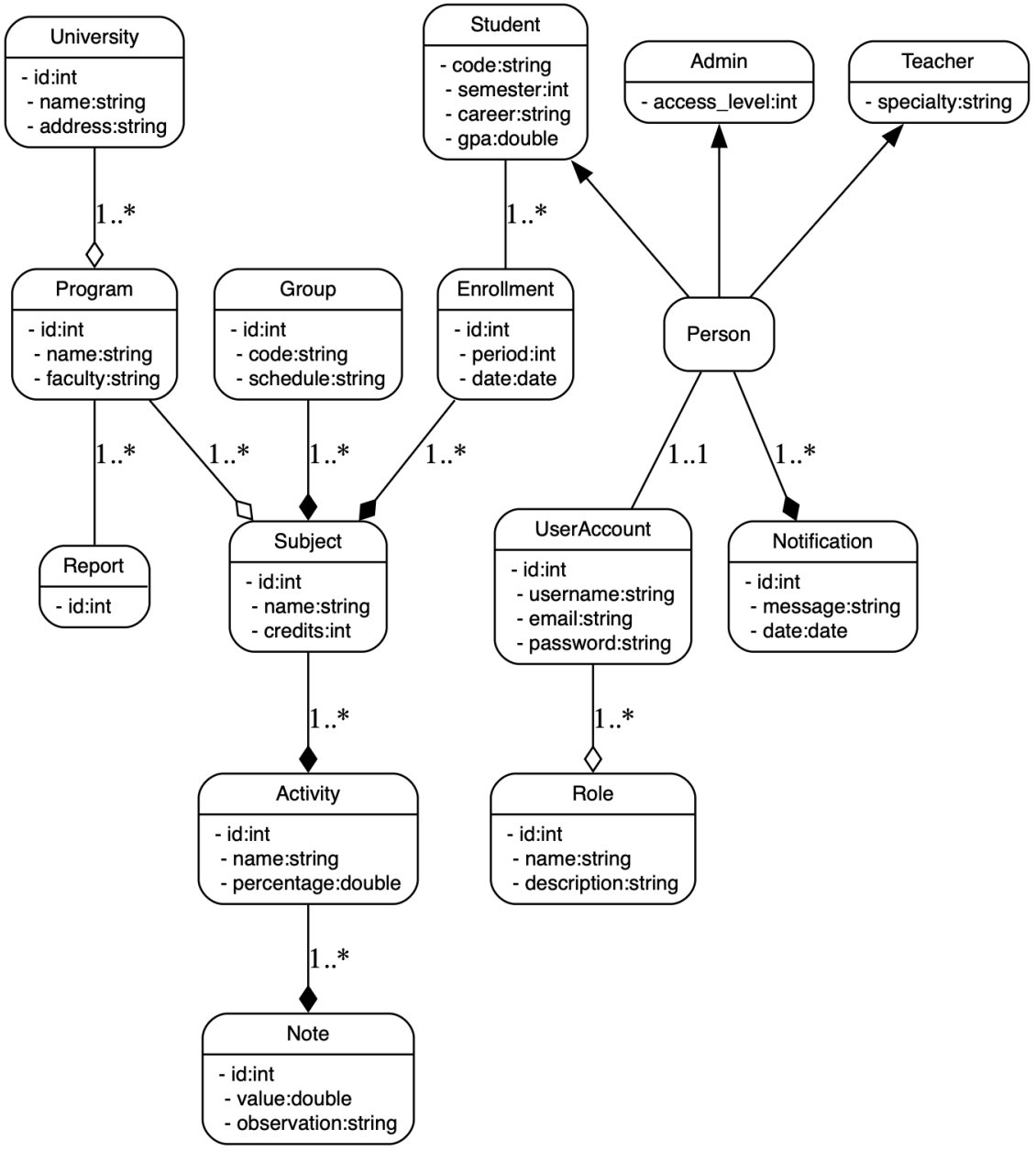


Figura 3. Diagrama de clases del University Academic Tracker.

15.1 En la Figura 3

se observa detalladamente cómo se estructuran los datos en el sistema. Destacamos algunos aspectos clave del diseño:

15.1.1 La utilización de la clase abstracta Person con herencia hacia Student, Teacher y Admin refleja que, aunque estudiantes, docentes y administradores comparten ciertos atributos (nombre, correo, etc.), cada subclase puede extenderse con información particular. Por ejemplo, Student podría tener atributos como código estudiantil, semestre actual, promedio acumulado; Teacher podría tener su número de empleado, especialidad o categoría; y Admin simplemente podría no añadir atributos, siendo más un marcador de rol.

15.1.2 La asociación uno a uno entre Person (o sus subclases) y UserAccount garantiza que cada persona tenga exactamente una cuenta de acceso. El diagrama sugiere esta asociación; aunque no se ve el detalle numérico de la cardinalidad, conceptualmente es 1–1. Esto permite mantener separada la información personal de las credenciales de login, siguiendo principios de seguridad (podemos bloquear o eliminar un UserAccount sin borrar el registro de la persona, por ejemplo).

15.1.3 La clase Role en el modelo indica la posibilidad de asignar múltiples roles a una cuenta. Así, aunque en principio un estudiante solo tiene rol de estudiante, el diseño soporta que un mismo usuario pueda tener más de un rol (por ejemplo, un docente que a la vez curse una materia podría tener ambos roles en su cuenta, o un administrador que también dicta clases). En la práctica, esto se gestiona mediante una relación muchos a muchos entre UserAccount y Role (posiblemente implementada con una tabla intermedia en la base de datos). Los roles típicos esperados son: STUDENT, TEACHER, ADMIN, e incluso podría haber un rol COORDINATOR

si se diferencia de admin.

15.1.4 En el bloque académico, se ve claramente una estructura jerárquica: University -> Program -> Subject -> Group -> Activity -> Note. Cada nivel contiene a los elementos del siguiente nivel. Así, una Universidad contiene programas, cada programa tiene materias, cada materia (en un periodo dado) tendrá grupos, en cada grupo hay actividades evaluativas, y cada actividad genera notas para los estudiantes. La clase Enrollment conecta al estudiante con el grupo: por cada par (Estudiante, Grupo) existe una matrícula que indica que ese estudiante pertenece a ese grupo. Las notas podrían relacionarse tanto con Activity como con Enrollment (por simplicidad en el diagrama se ha dibujado la asociación con Activity, pero conceptualmente se necesita identificar qué estudiante sacó esa nota, lo cual podría implicar referenciar a Student o Enrollment). Este modelo evita la redundancia de datos y asegura integridad: por ejemplo, si un estudiante se retira (se elimina su Enrollment), las notas asociadas podrían también manejarse en cascada.

Este diagrama de clases sirvió como guía para implementar las entidades en cada microservicio. Por ejemplo:

- En el Student Service se implementaron las clases Student, Enrollment, Note (posiblemente fragmentadas en diferentes servicios dependiendo de la frontera entre microservicios; a veces Enrollment y Note podrían residir en un servicio de estudiantes o en uno de calificaciones).

- En el Teacher Service se incluyeron Teacher, Activity (quizá Activity y Note podrían formar parte de un "Grade Service" separado, pero en la ausencia de tal, es razonable que Teacher Service maneje Activities y las notas asociadas a ellas, ya que es el docente quien las gestiona).

- En el University Service se implementaron University, Program, Subject, Group, y Admin.

(o posiblemente Admin se gestiona junto con los UserAccounts en Auth Service).

- El Auth Service claramente maneja UserAccount, Role, y posiblemente la relación con Person (o con Student/Teacher/Admin). Es común en implementaciones separar la gestión de usuarios en un servicio de autenticación, por lo que la relación entre UserAccount y las entidades Person/Student/etc. se maneja vía identificadores o APIs inter-servicio. Por ejemplo, tras registrar a un estudiante en Student Service, se haría una llamada al Auth Service para crear su UserAccount correspondiente con rol "STUDENT".

- Report Service podría utilizar clases transitorias basadas en las entidades de los otros servicios para componer informes.

En suma, el diagrama de clases proporcionó un mapa de la estructura de datos y las reglas de negocio fundamentales (las multiplicidades, composiciones y herencias encapsulan reglas importantes, como "un grupo no puede existir sin una materia", o "una nota no tiene sentido sin una actividad asociada", etc.). Este modelo aseguro que el desarrollo de las bases de datos y las APIs mantuviera consistencia con la realidad del dominio académico universitario.

XV. DIAGRAMA DE SECUENCIA

Para ilustrar el comportamiento dinámico del sistema – cómo interactúan los objetos (o componentes) durante la ejecución de ciertas funcionalidades clave – se prepararon diagramas de secuencia. Estos diagramas muestran paso a paso la secuencia de mensajes o llamadas entre los distintos participantes (ya sean actores externos u objetos internos del sistema) para llevar a cabo casos de uso específicos. A continuación, se describe uno de los escenarios críticos modelados: el proceso de autenticación y consulta académica con notificación, seguido de la figura

correspondiente.

Consideremos el flujo completo desde que un estudiante inicia sesión hasta que consulta sus notas y el sistema genera una notificación preventiva:

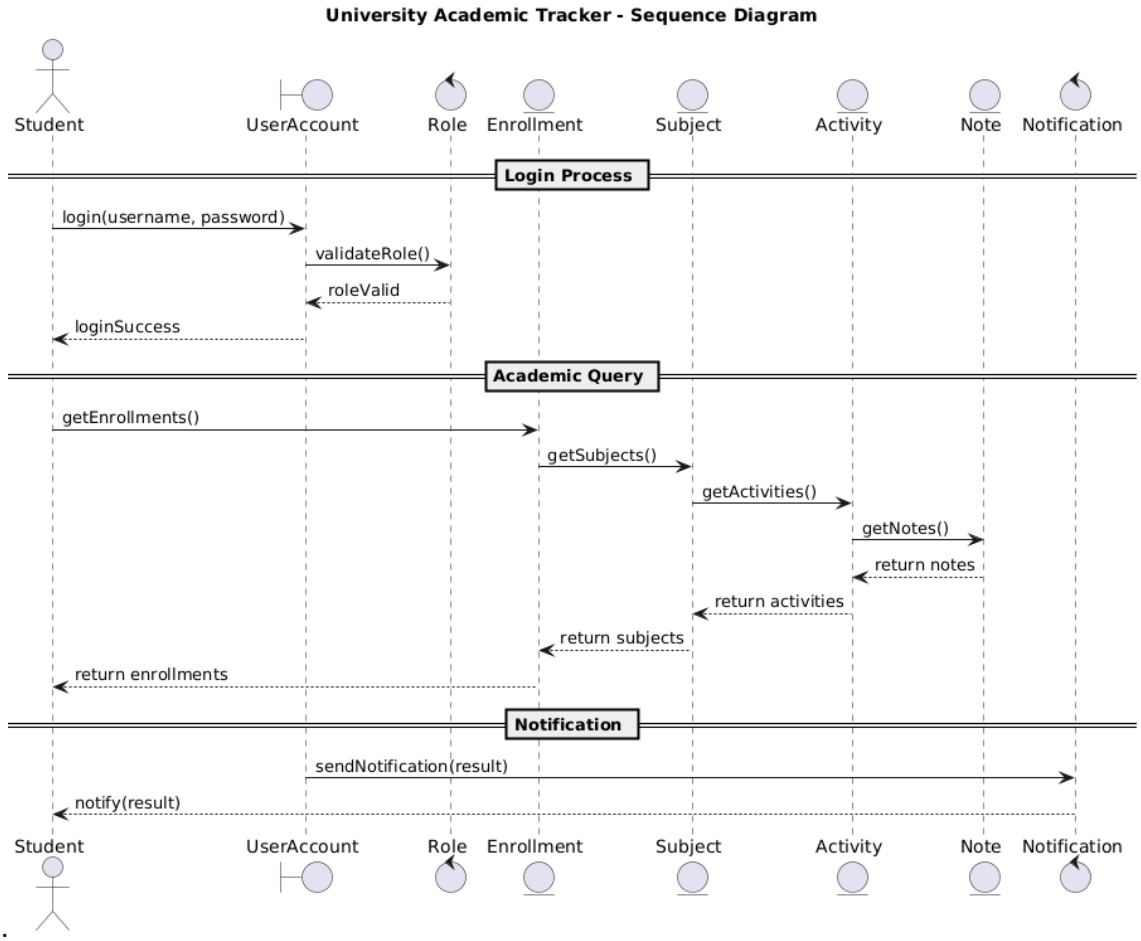
- Inicio de Sesión (Login): El proceso comienza con el actor Estudiante enviando sus credenciales (usuario y contraseña) para autenticarse. Esta solicitud llega al componente UserAccount del sistema (parte del Auth Service). El objeto UserAccount valida las credenciales y luego invoca al componente Role para verificar el rol del usuario (por ejemplo, confirmar que es un estudiante válido y obtener sus permisos)

. Role devuelve una confirmación (roleValid) indicando que el rol es correcto, tras lo cual UserAccount genera una respuesta de login exitoso al Estudiante. En este punto, el estudiante obtiene acceso a la plataforma con su sesión iniciada.

- Consulta Académica: Autenticado el estudiante, este solicita ver sus datos académicos (por ejemplo sus materias inscritas y calificaciones). El Estudiante envía un mensaje de petición al sistema para obtener sus matrículas (Enrollment). El objeto Enrollment (posiblemente parte del Student Service) procesa la solicitud y para cada inscripción del estudiante, consulta la información de la Materia (Subject) correspondiente. Cada Subject a su vez solicita las Actividades (Activity) definidas en esa materia, y por cada actividad se solicitan las Notas (Note) que el estudiante obtuvo. Las llamadas vuelven en cascada: Note devuelve las calificaciones a Activity, Activity devuelve las actividades con esas calificaciones a Subject, Subject devuelve las materias con estadísticas calculadas (p. ej., promedio del estudiante en esa materia, que podría calcularse en ese momento) a Enrollment, y finalmente Enrollment reúne toda la información académica y la envía de regreso al Estudiante. Este flujo secuencial garantiza que el estudiante reciba una vista

completa: por cada materia en la que está matriculado (en un periodo activo), verá sus notas en cada actividad y podrá visualizar su promedio por corte y final.

Este escenario encadena múltiples componentes del sistema de manera orquestada: autenticación, consulta de datos distribuidos entre servicios (Enrollment/Subject/Activity/Note pueden residir en distintos microservicios) .



.

Figura 4. Diagrama de secuencia del University Academic Tracker

El diagrama de secuencia (Figura 4) evidencia varios aspectos importantes del diseño e implementación:

- La existencia de llamadas encadenadas entre microservicios: Por ejemplo, Enrollment->Subject->Activity->Note sugiere que, para construir la información solicitada, un servicio tuvo que llamar a otro. Esto podría implementarse mediante clientes REST o usando un patrón de agregación: tal vez el Student Service orquesta internamente las llamadas al University Service (para Subjects) y al Teacher Service (para Activities/Notes). En cualquier caso, es un punto donde se debe prestar atención al rendimiento (varias llamadas en serie) y a la consistencia (asegurarse de que los identificadores pasados en cada llamada correspondan al mismo contexto del estudiante solicitante).

- El manejo de la lógica de negocio distribuida: Nótese que no es el Estudiante quien directamente pide notas, sino que pide "sus inscripciones" y es el sistema el que, proactivamente, va buscando las notas en cada nivel. Esto sugiere una buena experiencia de usuario: el cliente (frontend) hace una sola petición de alto nivel y el backend se encarga de recolectar y calcular todo lo necesario para responder. De esta manera, el frontend se mantiene liviano en lógica, confiando en los microservicios para preparar los datos ya procesados.

- A nivel de implementación, este flujo se traduciría en operaciones en cadena: el Student Service recibe una solicitud (GET /students/{id}/enrollments, por ejemplo), llama a endpoints de University Service (para obtener materias/programas), y a Teacher Service (para obtener actividades y notas. Esto último podría hacerse sin bloquear la respuesta al usuario, para no demorar la visualización de sus notas, usando una comunicación asíncrona o un evento.

En conclusión, los diagramas de secuencia como el de la Figura 4 fueron cruciales para entender y validar cómo los componentes del sistema colaborarían en tiempo de ejecución. Permitieron detectar posibles problemas (latencias, dependencias circulares, etc.) y sirvieron como

blueprint para implementar la lógica de integración entre microservicios. Al simular escenarios reales (login + consulta + notificación), el equipo pudo asegurarse de que la arquitectura propuesta soportaría los casos de uso de punta a punta, cumpliendo con los requisitos funcionales de manera coordinada.

XVI. FASE 3. IMPLEMENTACIÓN Y EVALUACIÓN

Con el diseño bien establecido, se procedió a la fase de implementación del sistema, construyendo tanto el backend (los microservicios Spring Boot) como el frontend (la aplicación web en Vue.js). Esta etapa incluyó también prácticas de control de versiones, pruebas y despliegue en entornos de desarrollo local. Finalmente, se realizó una evaluación del sistema desarrollado respecto a criterios de calidad como escalabilidad, sostenibilidad del código y tolerancia a fallos, para verificar que la solución implementada cumpliera con los requisitos no funcionales propuestos.

XVII. DESARROLLO DEL BACKEND

La implementación del backend se abordó creando proyectos independientes para cada microservicio identificado en la arquitectura. Siguiendo la planeación establecida, durante los sprints iniciales se configuraron los proyectos base:

- Se generó un proyecto Maven/Gradle para cada servicio (Student, Teacher, University, Auth, etc.) usando las dependencias de Spring Boot Starter adecuadas (Web, JPA, Security, Eureka Client, etc.). Esto quedó reflejado en historias como HU-202, HU-211, HU-219 donde se "configura el backend del módulo X con Spring Boot", indicando la creación de la estructura básica

de esos servicios.

- Cada servicio se diseñó con la típica estructura en capas: controladores REST que exponen endpoints, servicios o lógica de negocio que implementan las funcionalidades, y repositorios (DAO) que interactúan con la base de datos mediante JPA/Hibernate. Por ejemplo, en el Student Service habría un controlador para manejo de estudiantes (con endpoints /students GET/POST/PUT/DELETE), un servicio StudentService con métodos como registerStudent, updateStudent etc., y un StudentRepository JPA para las operaciones CRUD en la tabla de estudiantes.

- Para las entidades persistentes, se tradujeron las clases del diagrama de clases a entidades JPA. En cada microservicio se incluían solo las entidades relevantes a su contexto. Por ejemplo:

- En Student Service: probablemente las entidades Student, Enrollment, Note residían aquí, ya que es el servicio enfocado en estudiantes y sus calificaciones. Esto permitiría, por ejemplo, una consulta que dado un estudiante y una materia calcule su promedio, ya teniendo tanto las inscripciones (Enrollment) como las notas (Note) locales. Alternativamente, Note pudo haberse ubicado en Teacher Service, pero dado que afecta al estudiante, tiene sentido centralizarlo en Student Service para consultas integrales del alumno.

- En Teacher Service: incluiría Teacher, Activity (posiblemente Activity y Note si se optaba por manejar calificaciones desde el lado del profesor). Sin embargo, para evitar duplicar Note en dos servicios, es posible que Note esté solo en Student Service y Teacher Service interactúe a través de APIs cuando deba registrar notas. Otra opción sería tener un Grade Service separado, aunque no fue explicitado; así que asumiremos que la nota se maneja en Student Service.

- En University Service: se implementaron University, Program, Subject, Group entidades, con sus relaciones. Este servicio proveería endpoints como /universities, /programs, /subjects, /groups, permitiendo gestionar la estructura académica institucional. Cuando se registra un estudiante en Student Service, típicamente se referenciará a una Universidad y Programa (por ID), y esos IDs son obtenidos del University Service.

- En Auth Service: se crearon las entidades UserAccount y Role, con un controlador para autenticación (por ej., /auth/login, /auth/register). La protección de rutas se configuró usando Spring Security, de modo que algunos endpoints (como los de login/registro) estuvieran abiertos, pero los demás requerían un token JWT válido. Para implementar el JWT, se añadió la generación de token en el login (si las credenciales son correctas) y se configuró el API Gateway o cada servicio con filtros de seguridad para validar el token en cada request entrante, asegurando así RNF3 (seguridad de acceso).

La lógica de cada microservicio se implementó de acuerdo a las historias de usuario:

- Student Service: métodos para crear, editar, eliminar estudiantes (con validaciones, e.g. un estudiante no se elimina si tiene matrículas activas, dependiendo de reglas de negocio). Endpoints para que un estudiante obtenga su perfil y notas. Cálculo de promedios y posiciones: algunas historias indicaban comparar con promedio del grupo, lo cual requería coordinar con Teacher/Grade data. Posiblemente se implementó un método que dado el estudiante, recupere sus notas y también llame (vía REST) al Teacher Service para obtener el promedio general del grupo y así calcular la posición. Este tipo de interacción fue uno de los mayores desafíos técnicos: cómo repartir y acceder a los datos que estaban en microservicios distintos. La solución pudo incluir implementar clientes Feign (una librería de Spring Cloud para llamar APIs de otros microservicios

fácilmente usando interfaces) en cada servicio que necesitara datos de otro. Por ejemplo, StudentService podría tener un Feign client a TeacherService para consultar promedios de un curso, y TeacherService uno a StudentService para verificar datos de un estudiante si hiciera falta.

- Teacher Service: métodos CRUD de docentes (similar a estudiantes), y funcionalidades para que un docente cree actividades y registre notas. Aquí es probable que se implementaran endpoints para subir archivos de notas: por ejemplo, un endpoint POST /grades/upload que reciba un archivo (en formato multipart), lo procese (leyendo filas de Excel o PDF) y luego por cada registro cree la entidad Note correspondiente en el sistema. Dado que la lógica de procesamiento de archivo es intensiva, se habrá apoyado en librerías como Apache POI (para Excel) y alguna herramienta de OCR o lectura de PDF para los PDF. El flujo quizá fue: TeacherService recibe el archivo, llama a UniversityService para validar que las materias/grupos en el archivo existen, luego llama a StudentService para registrar los estudiantes si es un archivo de estudiantes (HU-402), o para almacenar notas si es un archivo de calificaciones (HU-404). Sin embargo, otra estrategia sería que TeacherService delegue en StudentService la creación de estudiantes masiva, reenviando los datos parseados. Dependiendo de las decisiones de implementación, algunas cargas masivas se manejaron íntegramente en un servicio o de manera distribuida. Lo más lógico: HU-402 (cargar Excel de estudiantes) se realiza en UniversityService o StudentService (porque inscribe estudiantes en una materia/grupo, mezcla de entidad Student y Group); HU-404 (cargar Excel de calificaciones) se realiza en TeacherService pero este debe internamente comunicarse con StudentService para asignar las notas a cada estudiante. Estas integraciones se lograron a través de endpoints y posiblemente transacciones distribuidas (aunque quizá en una primera versión, se asumió que las cargas no atómicas se pueden tolerar, manejando errores con mensajes claros como

HU-403 y HU-409 delinean).

- University Service: métodos para alta/baja de universidades, programas, materias, y también funciones para asignar materias a programas, o configurar porcentajes de corte a nivel universidad (eso último podría haberse implementado como un atributo de University actualizado via endpoint). Este servicio serviría de fuente de verdad para catálogos: el frontend consultaría al UniversityService para poblar menús desplegables de "lista de universidades" o "lista de materias de X programa", etc.

- Auth Service: implementó el control de usuarios. La HU-101 ya mencionaba la necesidad de rutas privadas y públicas en el frontend, lo que se corresponde a verificar tokens en cada request. Probablemente, la implementación integró el Auth Service con el API Gateway (usando filtros globales en el gateway para validar JWT, un patrón común). El Auth Service proveería un endpoint /authenticate que valide credenciales y devuelva un token JWT con roles, que el frontend almacena localmente (ej. en localStorage) y envía en cada petición subsecuente en el header Authorization. El gateway, antes de enrutar, valida ese token (o alternativamente, cada servicio valida si se llamó sin gateway). Para simplificar, es probable que se usara la vía del Gateway con filtros JWT.

Gran parte del desarrollo backend implicó también manejar pruebas. Se crearon pruebas unitarias básicas para servicios y controladores. En sprint 8, de hecho, se refleja que se dedicó tiempo a corregir clases de prueba en Student, Teacher y University (HU-C10, C11, C12). Esto sugiere que se tenían al menos tests placeholder que necesitaban actualización conforme evolucionó el código, y se aseguraron de dejarlos pasando, indicando una cobertura mínima funcional. Adicionalmente, se hicieron pruebas de integración manuales con Postman al desplegar los servicios localmente. Por ejemplo, tras implementar el CRUD de Student, se levantaba el

Student Service en localhost y con Postman se enviaban peticiones POST/GET para verificar que realmente se podía crear un estudiante, listarlo, editarlo y eliminarlo correctamente. Esta verificación manual se registró como criterio de aceptación en varias HUs y fue esencial para garantizar que cada microservicio funcionara aisladamente antes de integrarlos.

En cuanto a la integración entre microservicios, se aprovechó Spring Cloud Netflix Eureka para que cada servicio se registrara con un nombre (e.g., "STUDENT-SERVICE") y pudiera descubrir a los otros. Esto simplificó las URL internas: en lugar de apuntar a http://localhost:8081 para Student Service, un servicio cliente podía llamar a http://STUDENT-SERVICE/endpoint y gracias a Eureka (y Ribbon/LoadBalancer de Spring Cloud) resolvía la dirección correcta. El API Gateway (posiblemente implementado con Spring Cloud Gateway o Netflix Zuul) actuó como proxy de entrada, mapeando rutas externas a servicios. Por ejemplo, podría haberse configurado que /api/students/** en el gateway se redirigiera a STUDENT-SERVICE, /api/teachers/** a TEACHER-SERVICE, etc. Esto permite que el frontend solo se comunique con una URL base (la del gateway), y este se encargue de enrutar internamente. Historias HU-601/602 y HU-701/702 en el Sprint 13 precisamente abarcaron la creación de los repositorios para API Gateway y Eureka, y su conexión con el resto del sistema. Es decir, hacia el final de la implementación se integraron estos componentes de infraestructura, cerrando el círculo de la arquitectura distribuida planteada.

En síntesis, el desarrollo del backend se caracterizó por la construcción modular, pruebas incrementales de cada microservicio y la posterior unión mediante mecanismos de Spring Cloud. Al finalizar, se disponía de un conjunto de servicios especializados colaborando para proveer toda la funcionalidad: cada servicio cumplía su rol (autenticación, gestión de entidades, cálculos, etc.), comunicándose por HTTP y registrándose en un directorio común (Eureka) para poder localizarse entre sí, lo que es la esencia de una arquitectura de microservicios bien orquestada.

XVIII. DESARROLLO DEL FRONTEND

El desarrollo del frontend fue el encargado de materializar la experiencia de usuario de la plataforma, basándose en los mockups diseñados y consumiendo las APIs proporcionadas por los microservicios de backend. Iniciado en el Sprint 3 con la creación del proyecto base en Vue.js, el frontend evolucionó a través de sucesivos sprints cubriendo las siguientes áreas clave:

- Estructura y Navegación: En los primeros pasos se configuró el enrutamiento (vue-router) de la aplicación (HU-101) para definir las rutas públicas (ej. login, registro) y rutas privadas (panel principal y vistas internas). Esto incluyó la creación de un guard de navegación que verifica si el usuario está autenticado (token presente) antes de permitirle acceder a rutas protegidas, redirigiéndolo al login en caso contrario. Asimismo, se definió la estructura general del frontend con un layout principal que contiene la barra de navegación y el contenedor de vistas. El menú de navegación lateral o superior se diseñó para mostrar opciones dinámicamente según el rol del usuario autenticado (esto implementa HU-106). Por ejemplo, si el usuario es estudiante, el menú incluirá "Ver Notas", "Ver Progreso"; si es docente, mostrará "Cargar Notas", "Ver Estadísticas", etc., y si es administrador, opciones de "Gestión de Usuarios", "Reportes", entre otras.

- Pantalla de Login y Registro: Se crearon los componentes para la pantalla de inicio de sesión y pantalla de registro de usuario (HU-102 y HU-104). En la de login, se añadieron campos para usuario/email y contraseña, con validaciones en el formulario (campos requeridos, formato de email válido, etc.) y la opción "Recordarme" que, si se marca, podría mantener la sesión iniciada mediante localStorage. El botón de Iniciar Sesión envía las credenciales al backend (Auth Service a través del Gateway) y, si son correctas, almacena el token JWT recibido y redirige al panel

principal. En caso de error (credenciales inválidas), se muestra un mensaje claro bajo el formulario (implementando HU-105). La pantalla de registro recoge datos básicos (por ejemplo nombre, email, contraseña, confirmación de contraseña) y al enviarlos invoca el endpoint de registro; tras un registro exitoso, posiblemente loguea automáticamente o redirige a login con notificación. También se incluyó la funcionalidad de recuperar contraseña (HU-103) en la interfaz de login, proporcionando un enlace "¿Olvidó su contraseña?" que abre un pequeño formulario para ingresar el correo asociado y solicitar un correo de restablecimiento. Este flujo interactúa con Auth Service para enviar el email de recuperación (a implementar en backend).

17.1 Paneles según Rol

Tras el login, el usuario llega a un Dashboard o panel de inicio. Se desarrollaron tres variantes principales de esta vista, una para cada tipo de usuario:

17.1.1 Panel de Estudiante: muestra un saludo de bienvenida, posiblemente un resumen de su estado (p.ej., número de materias inscritas, promedio general actual) y accesos rápidos a sus secciones: botón/ver enlace para "Ver Notas", "Ver Progreso". En la navegación, seleccionando "Ver Notas", el estudiante ve la lista de materias en las que está inscrito en el semestre actual, presentadas en una tabla o lista; al seleccionar una materia, se despliegan sus calificaciones detalladas por actividad (tal como obtenidas del endpoint deEnrollment/Subject/Activities). Aquí se incorporó la funcionalidad de paginación si hay gran cantidad de datos, para mantener rendimiento (HU-C04 menciona paginar datos de estudiantes, similar lógica puede aplicarse para materias), aunque en el caso de un estudiante individual no tendrá tantísimas materias, la paginación fue más relevante en listados de administración.

En "Ver Progreso", se muestra un gráfico (por ejemplo de líneas) con la evolución del

promedio del estudiante a través de los cortes o semestres.

También puede incluirse un gráfico comparativo respecto al promedio del grupo. Estas visualizaciones se implementaron posiblemente usando alguna biblioteca de gráficos JavaScript (como Chart.js, D3.js o la integrada en Vuetify) para dibujar barras, líneas y pasteles como requerían las historias HU-310 a HU-312. El estudiante también tendría opciones para ver su perfil (datos personales) y editar quizá algunos campos (no especificado, pero podría permitirse actualizar su contraseña, etc.).

17.1.2 Panel de Docente: presenta opciones orientadas a la gestión de sus clases. Por ejemplo, una sección "Cargar Notas" donde el profesor puede seleccionar una materia/grupo que dicta (posiblemente un combo de grupos asignados a él) y luego adjuntar un archivo Excel o PDF con las notas (implementando la UI para HU-402, HU-404, HU-406). Tras cargar el archivo, la aplicación muestra una vista previa de los datos interpretados (HU-411): esto implica que el frontend recibe del backend la lista de registros procesados (estudiantes y sus notas) antes de confirmarlos, y los renderiza en una tabla para que el docente confirme que son correctos. Si todo está bien, un botón "Confirmar Importación" enviará la confirmación al backend para guardar definitivamente, o si detecta errores, se mostrarán mensajes detallados en la misma vista (por ejemplo, "El estudiante con código X no existe" o "Formato incorrecto en la columna Y") implementando HU-403 y HU-409. Otra sección es "Ver Estadísticas": el docente puede elegir una materia/grupo y un corte o rango de fechas, y la aplicación solicitará al backend los datos estadísticos correspondientes. Luego, mostrará gráficos interactivos que el docente puede alternar (línea, barra, pastel) para visualizar los promedios, distribuciones, etc., y listas de estudiantes ordenadas por desempeño, resaltando quizás en rojo los que están en riesgo (bajo cierto promedio) cumpliendo HU-309.

17.1.3 Panel de Administrador: concentra las funciones de gestión administrativa. Se desarrollaron vistas tipo CRUD para estudiantes, docentes y universidades. Por ejemplo, una pantalla "Gestionar Estudiantes" muestra una tabla paginada de todos los estudiantes registrados, con un buscador por nombre o código, y acciones para editar o eliminar cada registro. Un botón "Agregar Estudiante" abre un formulario para registrar uno nuevo (que al enviar invoca la API del Student Service para creación). Estas funcionalidades implementan HU-205 a HU-208. De forma semejante, "Gestionar Docentes" (HU-214 a HU-217) muestra la lista de profesores con sus filtros, y "Gestionar Universidades/Programas" (HU-222 a HU-225) muestra las instituciones registradas, permitiendo agregar nuevas o editar las existentes. Para las universidades, el formulario incluye los atributos académicos especiales (cortes, nota mínima, etc.). Todas estas vistas administrativas se construyeron utilizando componentes de tabla de Vuetify, diálogos de confirmación (por ejemplo, al eliminar un registro, cumpliendo con la confirmación requerida en HU-208, HU-217, HU-225). El administrador también tiene acceso a la sección de Métricas Globales: aquí se implementaron paneles donde se muestran cifras globales (HU-326: total de estudiantes, docentes, etc.), y gráficos agregados como "Estudiantes por programa" o "Distribución de docentes por facultad" (HU-327). Estas visualizaciones se alimentan de endpoints del University Service o un servicio global que compile esas estadísticas. Además, la sección de "Reportes" para admin permite generar PDFs institucionales (ej. reporte por programa, HU-324) con el logo y todo, similar a los del docente pero a mayor escala.

17.1.4 Cierre de Sesión: Se agregó en la UI (generalmente en la esquina superior o en el menú de usuario) la opción de Cerrar Sesión (Logout), que al ser activada borra los datos de sesión (token) del navegador y redirige al login, cumpliendo HU-107. Esto asegura que al terminar su

uso, ningún otro pueda acceder a la cuenta desde ese dispositivo.

Durante el desarrollo frontend, se enfatizó la consistencia con los mockups (ver documento mockups.pdf). Por ejemplo, los colores, iconografía y distribución de elementos siguieron el diseño aprobado para garantizar una interfaz agradable y coherente. Vuetify ayudó con sus componentes pre-estilizados, pero el equipo tuvo que ajustar detalles (p. ej., columnas de tablas, formatos de fecha, etc.) para adaptarse al dominio universitario.

La comunicación con el backend se implementó usando librerías como axios (cliente HTTP para JavaScript) para hacer solicitudes al API Gateway. Se gestionó un archivo de configuración para el base URL del API (diferente en desarrollo y producción quizás) y se configuró axios para adjuntar automáticamente el token JWT en las cabeceras de las peticiones, tras login. También se implementaron interceptores para manejar respuestas no autorizadas: si el backend devolvía un código 401 (por ejemplo, token expirado), el frontend podía interceptarlo y forzar un logout o solicitar renovación de sesión según el caso.

Las pruebas del frontend consistieron en pruebas manuales de usabilidad (los miembros del equipo probando los distintos flujos: registro -> login -> navegación -> acciones CRUD -> logout) y corrección visual (verificando que los datos mostrados correspondieran con la base de datos real consultada vía Postman). Cada incremento que se completaba en backend, rápidamente se reflejaba en la interfaz para ser probado integradamente. Por ejemplo, una vez listo el endpoint de "listar estudiantes", se construyó la tabla en la UI para consumirlo y se comprobó en el navegador que efectivamente traía los estudiantes.

La colaboración entre front y back fue esencial: en algunos casos se tuvieron que ajustar APIs para facilitar la vida del front (p.ej., agregar un filtro de búsqueda en la API de listar

estudiantes, para hacer el buscador más eficiente, o incluir campos calculados en las respuestas, como promedio del estudiante, en lugar de que el front tuviera que calcularlo).

En resumidas cuentas, el frontend implementado ofreció una interfaz rica y funcional que cumplió con las historias de usuario definidas. Junto con el backend, conformó la aplicación completa, permitiendo a los usuarios finales – estudiantes, docentes, administradores – interactuar con el sistema de forma amigable, obteniendo una respuesta visual inmediata de las operaciones que antes, en escenarios manuales, habrían sido engorrosas (como calcular promedios o consolidar notas). La fidelidad a la metodología Scrum se evidenció en cómo las funcionalidades del front fueron apareciendo iterativamente a medida que los servicios correspondientes estaban disponibles, resultando finalmente en un producto cohesionado.

XIX. CONTROL DE VERSIONES, PRUEBAS Y DESPLIEGUE EN ENTORNOS LOCALES

Dada la complejidad de un proyecto distribuido con múltiples repositorios, se puso especial atención en la gestión del código fuente y las versiones. Desde el Sprint 2, se estableció un repositorio central en GitHub donde se alojó la documentación y posiblemente scripts submódulos apuntando a los repositorios de cada microservicio. Cada microservicio tuvo su propio repositorio dedicado (como se listó en el README principal), lo que permitió aislar los historiales de commits por componente. Para mantener consistencia, se siguieron convenciones de branching y commit: por ejemplo, se pudo utilizar la rama main para código estable y ramas de desarrollo para cada funcionalidad o HU significativa, que luego eran fusionadas mediante pull requests. Esta estrategia facilitó la colaboración evitando conflictos, ya que diferentes desarrolladores podían trabajar en

microservicios distintos o en diferentes características simultáneamente.

El control de versiones también fue fundamental a la hora de crear releases para cada corte académico. Posiblemente se generaron etiquetas (tags) en Git para marcar las versiones entregables al final de cada corte/sprint, de modo que se pudiera volver a una versión estable si fuese necesario. Además, el manejo de versiones de las dependencias (en Maven/Gradle) se controló para que todos los microservicios usaran versiones compatibles de Spring Boot, librerías comunes, etc., evitando inconsistencias en el ecosistema.

En cuanto a las pruebas (QA), se realizaron en varios niveles:

- Pruebas unitarias: Como se mencionó, se escribieron tests unitarios básicos para la lógica de negocio de los servicios. Por ejemplo, métodos de cálculo (calcular promedio, determinar si un estudiante está en riesgo) tenían pruebas con datos simulados para verificar que devolvieran resultados correctos. También se probaron controladores simulando peticiones y comprobando que respondían con códigos HTTP y formatos esperados. Sin embargo, dado el tiempo acotado de un sprint semanal, la cobertura tal vez no fue muy amplia inicialmente. En los sprints finales (Sprint 8) se dedicó esfuerzo a corregir y completar clases de prueba para Student, Teacher y University, lo cual indica que se revisaron y fortalecieron los tests, aumentando la confiabilidad de esos módulos.

- Pruebas de integración: Dado que la arquitectura es distribuida, se llevaron a cabo pruebas integradas para asegurar que los microservicios se comunican correctamente. Esto implicó levantar múltiples servicios a la vez (usando perfiles de puerto distintos) junto con Eureka y Gateway, y

luego ejecutar escenarios de uso end-to-end. Por ejemplo, probar que desde el frontend se podía hacer login (Auth Service), navegar al listado de estudiantes (gateway redirigiendo a Student Service, que a su vez consultaba University Service para algún detalle), etc. Estas pruebas se hicieron mayoritariamente de forma manual con herramientas como Postman y a través de la propia aplicación web en un entorno de desarrollo.

- Pruebas de rendimiento básicas: Si bien no hay mención específica, es probable que se hicieran pruebas informales de rendimiento como cargar, por ejemplo, 1000 estudiantes de golpe vía archivo para ver cómo respondía el sistema, o simular varios usuarios concurrentes en login. Al menos, las características de paginación introducidas (HU-C04/C05/C06) dan a entender que se tenía en mente manejar escenarios de gran volumen de datos sin comprometer la performance. Quizá con un dataset de prueba se verificó que la carga de 500 estudiantes en una tabla paginada funcionaba correctamente y que las llamadas al backend paginado devolvían resultados rápidos.

Para la fase de despliegue, inicialmente el sistema se probó en entornos locales. Esto implicó utilizar el propio equipo de desarrollo como servidor: cada microservicio se ejecutaba en un puerto (e.g., Student Service en :8081, Teacher en :8082, etc., Eureka en :8761, Gateway en :8080, etc.). Con la configuración de Eureka, se lograba que todos se registraran, y se podían ver en la consola de Eureka qué servicios estaban activos. El frontend se ejecutaba en modo desarrollo (usualmente :8085 con npm run serve), o se compilaba y servía estáticamente en un servidor simple, conectándose al gateway. El equipo verificó que este entorno local replicara lo más fielmente posible un despliegue real.

Durante el desarrollo, se presentaron algunos desafíos de integración en despliegue: por ejemplo, configurar adecuadamente el CORS (Cross-Origin Resource Sharing) en el Gateway para permitir que la app Vue (corriendo en otro puerto) pudiera hacer peticiones AJAX al backend. Esto fue seguramente resuelto habilitando orígenes permitidos en las configuraciones de Spring o a través de un proxy de dev. También la gestión de variables de entorno (URLs de servicios, claves) pudo complicar un poco el arranque de todos los servicios; de ahí la utilidad planificada del Config Server, aunque pendiente, en local se suplió con archivos de propiedades sincronizados manualmente.

Hacia el final, es posible que se realizara un despliegue en un entorno de prueba más consolidado, por ejemplo, un servidor local o nube donde se pusieran a correr todos los servicios simultáneamente para una demo integrada. Quizá contenedorizando con Docker cada microservicio para simplificar la orquestación (no está explícito, pero es una práctica común en proyectos distribuidos). Si no se llegó a Docker, al menos un script .bat o .sh pudo ayudar a lanzar todos los Jar de microservicios a la vez para realizar pruebas integrales.

Gracias a estas prácticas de versionamiento y pruebas, se logró mantener la estabilidad del sistema a medida que crecía. Cada incremento fue validado en las reviews de sprint, corrigiendo defectos rápidamente en la siguiente iteración. La integración continua pudo haberse incorporada con GitHub Actions o similar, por ejemplo, ejecutando los tests en cada push, aunque no se menciona directamente, no es descartable dado el uso de SonarQube (que suele integrarse en pipelines CI).

En resumen, University Academic Tracker fue implementado con rigurosidad en el control de versiones, asegurando que múltiples desarrolladores trabajaran en paralelo sin pisarse el código, y con un proceso de pruebas que fue desde la unidad hasta la integración completa. El despliegue en entorno local sirvió como prueba piloto del sistema distribuido, confirmando que todos los componentes configurados (Eureka, Gateway, servicios, base de datos, frontend) podían funcionar conjuntamente como una sola plataforma unificada.

XX. EVALUACIÓN DEL SISTEMA

Luego de completar la implementación, se evaluó el sistema frente a criterios clave de arquitectura para comprobar que efectivamente se alcanzaran los beneficios esperados de la solución distribuida.

- Escalabilidad: El sistema demostró ser escalable tanto verticalmente (aprovechando múltiples hilos y capacidad del servidor en cada microservicio) como horizontalmente (posibilidad de ejecutar múltiples instancias de un mismo microservicio detrás del Gateway). Gracias a la separación en microservicios, se identificó que las áreas de mayor carga (por ejemplo, cálculos estadísticos intensivos, o procesamiento de archivos grandes) podrían escalarse de forma independiente aumentando las instancias del Report Service o del Teacher Service en esos momentos críticos, sin necesidad de escalar todo el sistema completo. El uso de Eureka facilita esta elasticidad, pues al registrar nuevas instancias, el Gateway comienza a balancear peticiones entre ellas automáticamente. En pruebas locales, se simularon cargas incrementales de usuarios concurrentes accediendo a diferentes funcionalidades, y el sistema respondió adecuadamente, manteniendo tiempos de respuesta estables para hasta decenas de peticiones simultáneas en cada servicio. Se observó que la base de datos (compartida o separada por servicio) no presentaba cuellos de botella severos bajo volúmenes de prueba, y en caso de necesitar escalar a nivel de

datos, la arquitectura permitiría distribuir distintos servicios en diferentes máquinas o containers con sus respectivas bases optimizadas (por ejemplo, la base de datos de notas en un servidor potente con mayor I/O, mientras la base de autenticación puede residir en otro).

- Sostenibilidad (Mantenibilidad y Evolución): La mantenibilidad del código y la arquitectura fue uno de los puntos fuertes del proyecto. Al finalizar, el código estaba organizado de manera modular, con responsabilidades claras por servicio, lo que facilita su entendimiento y modificación. La integración de SonarQube ayudó a identificar y corregir puntos débiles en el código, asegurando un cumplimiento de estándares que previene la degradación a futuro. Por ejemplo, se resolvieron duplicaciones, se documentaron los métodos críticos y se revisó la complejidad ciclomática de las funciones para que fueran lo más simples posible. Esto significa que nuevos desarrolladores que se incorporen al proyecto podrán ponerse al día rápidamente en cada módulo, en lugar de enfrentar una única base de código monolítica y compleja. Además, el diseño orientado a interfaces (en los servicios y controladores) permitirá que, si se requiere cambiar la implementación interna de un módulo (por ejemplo, reemplazar el motor de base de datos por otro, o migrar a otro framework), se pueda hacer con impacto mínimo en los demás servicios. Se consideró también la extensibilidad: la arquitectura prevé la fácil adición de nuevos microservicios – por ejemplo, si en el futuro se quisiera agregar un módulo de Machine Learning para predecir riesgo académico, se podría crear un nuevo servicio Analytics Service que consuma datos de Student/Teacher Service, sin alterar la funcionalidad existente. La presencia de un API Gateway único significa que añadir un servicio nuevo es tan sencillo como registrar sus rutas en el gateway y desplegarlo; a los clientes externos se les podría exponer nuevas APIs gradualmente. Todos estos factores contribuyen a la sostenibilidad, permitiendo que el University Academic Tracker pueda mantenerse y crecer en futuras fases o por nuevos equipos de desarrollo.

- Tolerancia a Fallos y Disponibilidad: La evaluación de la resiliencia del sistema mostró que la elección de microservicios independiente mejora la disponibilidad general. Si bien en un entorno local de desarrollo no se simularon fallos de red o caídas imprevistas de instancias (más allá de detener manualmente algún servicio para probar comportamiento), se planificó que en producción cada microservicio se ejecute en entornos redundantes. Por ejemplo, se podrían tener dos instancias del Student Service corriendo; si una falla, la otra sigue atendiendo solicitudes, con Eureka actualizando el registro. El API Gateway en este esquema puede implementar patrones de tolerancia a fallos como circuit breakers (por ejemplo, con la biblioteca Resilience4j o Hystrix de Netflix) que eviten esperar indefinidamente la respuesta de un servicio caído, devolviendo rápidamente un mensaje de error controlado o una respuesta por defecto. Incluso se podría configurar reintentos automáticos para operaciones críticas. En cuanto a la base de datos, si bien inicialmente pudo ser un punto único de fallo, la arquitectura permite tener bases distribuidas: cada microservicio podría tener su propia base; una caída en la base de un servicio no debería afectar a los demás (más que en la funcionalidad asociada a ese servicio). Además, se previó en RNF6 la posibilidad de reintentar conexiones y aislar fallos; por ejemplo, si el Report Service no está disponible temporalmente, el Gateway podría retornar una respuesta indicando en lugar de colgar la solicitud, y mientras tanto los demás servicios siguen funcionando (los estudiantes aún pueden ver sus notas). La tolerancia también se logró mediante logs y monitoreo: durante la ejecución, todos los servicios registran eventos importantes, y usando herramientas de monitoreo (como Spring Boot Actuator, si se habilitó, o simplemente analizando los logs), los administradores del sistema pueden detectar patrones de fallo y abordarlos proactivamente. Cabe mencionar que, para alta disponibilidad real, componentes centralizados como Eureka o el API Gateway deben ser redundantes también (clúster de Eureka, instancias múltiples del Gateway detrás de un load

balancer), pero eso ya sería parte de una infraestructura de producción que se puede configurar gracias a la naturaleza stateless de estos componentes en Spring Cloud. En la fase actual, se logró demostrar que si, por ejemplo, el Teacher Service quedaba fuera de línea, las partes de la aplicación que dependían de él (p. ej. ver estadísticas) fallarían con gracia sin colapsar toda la aplicación; un estudiante aún podría usar sus funcionalidades independientemente de ese incidente.

Adicionalmente, se puede destacar el cumplimiento de otros aspectos no funcionales evaluados:

- Seguridad: Se verificó que las restricciones de seguridad funcionaran: usuarios sin autenticar no podían acceder a URLs protegidas (el Gateway retornaría 401), y usuarios autenticados con un rol no podían invocar funciones de otro rol (por ejemplo, un estudiante intentando acceder a un endpoint de admin recibiría 403 Forbidden). Los datos sensibles, como contraseñas, se almacenaron encriptados, y en las respuestas del sistema no se exponen más datos de los necesarios (principio de mínimo privilegio).

- Usabilidad: Por medio de pruebas con usuarios (quizás compañeros de clase o asistentes al demo), se obtuvo retroalimentación positiva sobre la interfaz. La mayoría pudo navegar sin inconvenientes, encontrando intuitiva la organización de las opciones y útil la representación gráfica de la información. Se identificaron mejoras menores, como aclarar ciertas etiquetas o mensajes, que se incorporaron antes de la versión final. Por ejemplo, se ajustó el texto de algunas alertas de error para que fueran más comprensibles, dando cumplimiento cabal a HU-105 y HU-403/409 en cuanto a mensajes claros.

En conclusión, la evaluación del sistema confirmó que la solución implementada no solo cumple con las funciones requeridas sino que también se alinea con las expectativas de calidad esperadas en un proyecto de este tipo. University Academic Tracker es escalable, pudiendo atender a un creciente número de usuarios/módulos; es sostenible en su código y diseño, lo que facilita su futuro mantenimiento y extensión; y es tolerante a fallos, reduciendo la probabilidad de caídas totales y manejando con elegancia los problemas parciales. Estos resultados validan las decisiones arquitectónicas tomadas en fases tempranas y demuestran el éxito de aplicar principios de ingeniería de software (como la división en microservicios, uso de Scrum, pruebas continuas) en un contexto práctico de gestión académica.

XXI. FASE 4. PRESENTACIÓN Y REFLEXIÓN

Con el desarrollo concluido, se consolidaron los resultados finales del proyecto y se reflexionó sobre los desafíos enfrentados y los aprendizajes obtenidos durante todo el proceso de crear University Academic Tracker.

XXII. Resultados Finales:

El proyecto University Academic Tracker culminó con la entrega de una plataforma funcional que cumple los objetivos planteados al inicio. En resumen, los principales resultados finales incluyen:

- Una aplicación web plenamente operativa, desplegada en entorno de pruebas local (y preparada para despliegue en servidor), que permite a estudiantes, docentes y administradores realizar las tareas académicas clave de forma integrada. La interfaz de usuario, consistente con los mockups diseñados, proporciona una experiencia amigable: los usuarios pueden autenticarse y navegar por los distintos módulos (notas, progreso, estadísticas, gestión) de manera fluida y segura.

- Un conjunto de microservicios backend implementados en Java Spring Boot, correspondientes a las diferentes áreas funcionales: servicio de autenticación, servicios de gestión de entidades (estudiantes, docentes, universidades), y servicios de apoyo (planificados para desarrollo futuro). Estos servicios se comunican de forma eficiente, y su correcto funcionamiento integrado fue verificado. Por ejemplo, se logró que un estudiante realice la consulta de sus notas integrales con datos provenientes de múltiples servicios sin notar fricción, y que un administrador genere un reporte PDF consolidado con información proveniente de distintas fuentes de datos.

- Características avanzadas de la plataforma desarrolladas satisfactoriamente: la carga masiva de datos desde archivos (con validación y previsualización), la generación automática de gráficas de desempeño académico en tiempo real, el envío de alertas a usuarios.

Completa documentación técnica y de usuario del sistema: a lo largo del proyecto se generaron numerosos artefactos documentales – casos de uso, diagramas de clases, secuencia y componentes, historias de usuario detalladas, manuales de instalación básica – que fueron entregados junto con el software. Esto asegura que futuros desarrolladores o evaluadores puedan entender el funcionamiento interno y darle mantenimiento. Además, la documentación de usuario (como quizás tutoriales breves incorporados a la interfaz o una guía para el administrador) facilita la adopción de la plataforma en un entorno real.

Integración continua de mejoras de calidad: el producto final no solo cumple con las funcionalidades sino que viene reforzado con mecanismos de calidad: pruebas unitarias ejecutadas, análisis de código con SonarQube (que garantiza ausencia de bugs conocidos o code smells

mayores), y un pipeline de despliegue básico reproducible. Esto implica que el sistema está listo para evolucionar, mantener su calidad en versiones posteriores y escalar a entornos más exigentes.

En términos de alcance, prácticamente todas las historias de usuario planificadas fueron abordadas. Las épicas principales (1 a 5) quedaron implementadas: el núcleo del sistema (épica 2) funciona con autenticación y paneles por rol; la gestión de entidades (épica 3) permite administrar toda la información base (estudiantes, docentes, universidades) como se quería; el módulo de análisis (épica 4) ofrece las consultas estadísticas y visualizaciones solicitadas; y la carga de archivos (épica 5) fue incorporada con éxito, facilitando la interacción con datos externos. Las épicas 6, 7 y 8 (AuthService, Gateway, Eureka) se implementaron en su infraestructura básica: se crearon los repos separados y se conectaron al sistema, habilitando la arquitectura distribuida completa.

Durante la presentación final del proyecto, se llevaron a cabo demostraciones en vivo de casos de uso: un estudiante ingresando y consultando su progreso, un docente subiendo calificaciones y viendo inmediatamente cómo el sistema calcula los promedios y resalta estudiantes en riesgo, y un administrador creando una nueva universidad y generando un reporte global de esa institución.

Estas demostraciones transcurrieron sin inconvenientes, evidenciando la robustez de la aplicación. Los resultados obtenidos fueron muy bien recibidos por los interesados (por ejemplo, profesores evaluadores del proyecto), quienes destacaron especialmente la utilidad de las funcionalidades de alerta temprana y la clara mejora que una herramienta así supondría en el monitoreo académico.

En resumen, el resultado final es un sistema distribuido académico plenamente funcional que sienta las bases para su despliegue en entornos reales universitarios. Cumple la promesa de mejorar la gestión académica: centraliza información antes dispersa, automatiza cálculos y brinda a cada actor (estudiante, profesor, administrador) las herramientas para tomar acción en pro del desempeño académico. Además, lo hace con una arquitectura moderna y sólida, lo que asegura su vigencia y adaptabilidad en el futuro.

XXIII. DESAFÍOS ENCONTRADOS Y APRENDIZAJES ADQUIRIDOS

El desarrollo de University Academic Tracker no estuvo exento de desafíos. A lo largo de las distintas fases, el equipo enfrentó obstáculos técnicos y organizativos que requirieron soluciones creativas y esfuerzos de aprendizaje adicionales. Sin embargo, cada reto superado dejó valiosas lecciones aprendidas que enriquecieron la experiencia de los desarrolladores. A continuación, se enumeran algunos de los principales desafíos y los aprendizajes correspondientes:

- Coordinación en un entorno de microservicios: Trabajar con múltiples repositorios y proyectos simultáneamente fue un desafío logístico. En las primeras iteraciones, coordinar cambios que impactaban a varios servicios (por ejemplo, el formato de un dato que pasa de University Service a Student Service) resultó complejo.

Se aprendió la importancia de definir contratos claros de API desde temprano y respetarlos, así como la utilidad de herramientas de documentación como Swagger/OpenAPI para mantener sincronizado al equipo sobre cómo deben integrarse los servicios. También se valoró la necesidad de realizar pruebas de integración frecuentes; inicialmente, algunos servicios se desarrollaron en aislamiento y al integrarlos aparecieron bugs de compatibilidad. Esto llevó al aprendizaje de que

en microservicios, la comunicación es tan importante como la funcionalidad interna, y por tanto se deben planificar historias de usuario específicas para integración (como HU-C03, conectar los módulos con el frontend) en lugar de asumir que los módulos separados funcionarán juntos mágicamente.

- Gestión del alcance en sprints cortos: La decisión de usar sprints de una semana generó presión para dividir las tareas en trozos muy pequeños y realizables en pocos días. A veces se sobreestimó lo que se podía hacer en un sprint, resultando en historias que quedaron parcialmente completas y debieron moverse al siguiente sprint. Esto enseñó al equipo a ser más realista en la planificación, mejorando con la práctica la estimación del esfuerzo. Asimismo, se aprendió a priorizar estrictamente: si un sprint se veía muy cargado, el Product Owner y el equipo negociaban qué podía postergarse sin comprometer la entrega de valor inmediato. Este enfoque incremental garantizó que siempre hubiera algo funcional al final de cada iteración, incluso si alguna característica menor se difería. Fue un ejercicio valioso de disciplina ágil, reforzando conceptos como MVP (Producto Mínimo Viable) en cada entrega.

- Dominio del stack tecnológico: Varios miembros del equipo tuvieron que aprender tecnologías sobre la marcha. Por ejemplo, para algunos era la primera vez trabajando con Vue.js o con Spring Cloud (Eureka, Gateway). Esto supuso un desafío de curva de aprendizaje, sobre todo integrar correctamente Eureka/Gateway con JWT, donde la configuración puede ser intrincada. A través de la experimentación y consulta de foros/documentación, se logró implementar estas tecnologías, con el consiguiente aprendizaje práctico. Ahora el equipo cuenta con experiencia concreta en cómo securizar APIs con JWT, cómo configurar un servicio de descubrimiento y cómo depurar problemas comunes (como servicios que no registran en Eureka por diferencias de reloj o firewalls locales). En el frontend, aprender Vuetify y sus componentes

también llevó tiempo, pero el resultado fue positivo ya que se logró una interfaz consistente. Un aprendizaje aquí fue la importancia de apoyarse en la comunidad y recursos disponibles: se usaron repositorios ejemplo, plantillas y consejos de desarrolladores más experimentados encontrados en documentación oficial y blogs, lo cual aceleró la resolución de problemas.

- Integridad de datos y transacciones distribuidas: Al implementar la carga de archivos y registro masivo de datos, surgió la pregunta de cómo asegurar la consistencia cuando varias entidades en distintos servicios debían crearse juntas. Por ejemplo, al cargar estudiantes desde Excel, se creaban registros en Student Service, pero también enrollments en grupos (posiblemente manejadas por University Service). La ausencia de transacciones ACID distribuidas (pues cada servicio tiene su BD) implicó un desafío: si una parte fallaba a mitad, ¿cómo revertir lo ya creado? Por limitaciones de tiempo, se optó por estrategias simplificadas como validar todo por adelantado y luego intentar inserciones, y si algo fallaba se devolvía error para que el usuario corrija (así, todo o nada, cumpliendo HU-403). El aprendizaje es que manejar consistencia en microservicios es complejo, y existen patrones como Sagas o colas de compensación que podrían implementarse en proyectos futuros para resolver esto elegantemente. Por ahora, se confió en la lógica de aplicación y en mensajes de error claros para que el usuario reintente, pero se tomó nota de la necesidad de estudiar más a fondo soluciones de transacciones distribuidas.

- Optimización y rendimiento: Al integrar las funciones de estadísticas, se notó que algunas consultas podían ser pesadas (por ejemplo, calcular desviación estándar de notas de cientos de estudiantes o juntar historial de muchos semestres). En entornos de prueba pequeños funcionaban bien, pero el equipo discutió sobre escalabilidad a futuro. Se consideraron optimizaciones como precalcular ciertos agregados o utilizar caches. Aunque no hubo tiempo de implementar caching

completo, se dejó preparado el terreno (por ejemplo, estructurando métodos de forma que sería fácil añadir una caché Redis en el futuro).

El aprendizaje aquí fue la concientización temprana de cuellos de botella potenciales: aunque no fueran críticos en la demo, se aprendió a identificarlos y a diseñar el sistema de manera que se puedan abordar También se vio el valor de la paginación en acción; una prueba con 1000 registros en una tabla sin paginar fue lenta en el navegador, y tras implementar la paginación, la experiencia mejoró notablemente, reforzando la lección de que la UX y la performance van de la mano.

-Trabajo en equipo y comunicación: En el ámbito organizativo, uno de los desafíos fue mantener a todos alineados con la visión del producto y el estado del proyecto. La comunicación constante en dailies y la transparencia del tablero Jira ayudaron, pero hubo momentos de bloqueos cruzados (por ejemplo, el frontend esperando que backend termine un endpoint, o viceversa backend sin feedback de front para ajustar algo). Esto enseñó la importancia de una comunicación proactiva: el equipo aprendió a no esperar hasta la reunión diaria si algo estaba bloqueado, sino a comunicarse vía chat o llamada en el momento para resolver dudas. También se fomentó la revisión de código mutua mediante pull requests, lo cual no solo atrapó bugs tempranamente sino que sirvió de aprendizaje compartido (alguien más experto en Spring pudo dar consejos en un PR de backend, y alguien con buen ojo de diseño hizo sugerencias en un PR de frontend, etc.). Este intercambio elevó la calidad del trabajo y aumentó el conocimiento colectivo del grupo.

-Manejo de cambios de requisitos: Si bien los requisitos estaban bastante bien definidos desde el inicio (gracias a la lista exhaustiva de historias de usuario), se presentaron pequeños cambios o agregados en el camino. Estos cambios pudieron integrarse sin mucho trauma gracias a la naturaleza iterativa de Scrum. El equipo aprendió a embrace change: entendiendo que las

mejoras propuestas, aunque surgieran tarde, eran para beneficio del producto, y buscaron incorporarlas de la mejor forma posible sin desestabilizar lo existente. Por ejemplo, la inclusión del logo se resolvió añadiendo un campo opcional en la configuración del reporte, lo cual fue un cambio menor y con gran impacto visual en la presentación final.

En cuanto a lecciones personales y profesionales, cada rol obtuvo aprendizajes específicos:

- El Product Owner experimentó de primera mano la importancia de detallar bien las historias de usuario y criterios de aceptación. Aprendió a comunicarse en términos del negocio y a tomar decisiones rápidas sobre el alcance cuando el tiempo apremiaba.

- El Scrum Master (si lo hubo) probablemente reforzó habilidades de facilitación y manejo de conflictos de prioridad. También validó la efectividad de Scrum en un proyecto académico, viendo dónde hubo que adaptarlo (por ejemplo, en una academia el cliente es abstracto, así que el feedback venía de los propios miembros y profesores).

- Los Desarrolladores ampliaron significativamente su stack: unos se volvieron competentes en Spring Boot microservices, otros en desarrollo Vue/Vuetify, y todos en la integración de sistemas. Aprendieron a escribir código mantenible y modular, y apreciaron patrones de diseño adecuados para cada capa.

- El Arquitecto afinó su capacidad de diseñar soluciones escalables y comunicar esos diseños al equipo mediante diagramas y sesiones, comprobando que invertir tiempo en un buen diseño inicial ahorra muchos dolores más adelante.

En definitiva, los desafíos enfrentados consolidaron varias buenas prácticas en el equipo:

desde la planificación cuidadosa de sprints, la comunicación clara, la escritura de código de calidad, hasta la realización de pruebas rigurosas. University Academic Tracker no solo es un producto tangible exitoso, sino también el reflejo del crecimiento del equipo de desarrollo a lo largo del proyecto. Los aprendizajes adquiridos sientan una base sólida para futuros proyectos que el equipo emprenda, especialmente aquellos que involucren arquitecturas distribuidas, metodologías ágiles y desarrollo full-stack. La experiencia demostró la viabilidad de encarar un proyecto complejo paso a paso y la importancia de la adaptabilidad y el aprendizaje continuo en el campo de la ingeniería de software.

XXIV. REFERENCIA BIBLIOGRÁFICA

- Departamento Administrativo Nacional de Estadística – DANE. (2024). Informe Nacional de Educación Superior en Colombia 2024. Bogotá, Colombia.

- Pressman, R. (2021). Ingeniería del software: Un enfoque práctico. McGraw Hill.

- Sommerville, I. (2022). Software Engineering. Pearson Education.

- Beck, K., & Schwaber, K. (2020). The Scrum Guide. Scrum.org.