Universidad Nacional de Mar del Plata
Facultad de Ingeniería
Departamento de Informática

G.I.A.

Sistema de Gestión de Instrumentos A

Autores:

Barriga, Nahuel
nahuel.barriga.nb@gmail.com

Firmani, Gregorio
gregoriofirmani@gmail.com

Trinitario, Bruno
brunotrinitario@gmail.com

Director: Spinelli, Adolfo Tomas

Referente funcional: Rico, Carlos Alberto

Proyecto final para optar al grado de Ingeniero en Informática

Mar del Plata, Diciembre de 2025

1. Agradecimientos....8
2. Resumen del Proyecto....8
3. Introducción....8
3.1. Objetivos del proyecto....9
4. Planificación del proyecto....10
4.1. Análisis del problema....10
4.2. Equipo de trabajo....11
4.3. Metodologías de trabajo....12
4.4. Análisis FODA....14
4.5. Solución planteada....15
4.5.1. Beneficios esperados....15
4.5.2. Alcance funcional....16
4.5.3. Arquitectura tecnológica....16
4.6. Análisis de riesgos....17
4.7. Estimación de tiempos (Gantt)....19
Fase I: Estimación Inicial Pre-Desarrollo....19
Fase II: Refinamiento Post Inicio del Desarrollo Técnico....21
5. Ejecución del proyecto....22
5.1. Análisis del proyecto....22
5.1.2. Modelado del sistema....23
5.1.3. Modelo de casos de uso....27
5.1.4. Modelado de procesos del sistema....31
5.1.5. Análisis de seguridad y permisos....33
5.2. Diseño del sistema....36
5.2.1. Diseño de módulos....37
5.2.2. Tecnologías....38
5.2.3. Diseño del despliegue del sistema....39
6. Desarrollo del proyecto....39
6.1. El Backend....40
6.1.1. División de trabajo....40
6.1.2. Etapas del desarrollo....41
Primer etapa....41
Segunda etapa....41
6.1.3. Soluciones implementadas....42
6.2. El Frontend....49
6.2.1. División de trabajo....49
6.2.2. Etapas del diseño....50
6.2.3. Etapa de desarrollo....52
6.2.4. Soluciones implementadas....52
6.3. Integración....55
6.3.1. Descripción general....55
6.3.2. Estrategia de integración....55

6.3.3. Proceso de integración..... 56
6.3.4. Pruebas end-to-end..... 56
6.3.5. Distribución de responsabilidades..... 56
6.3.6. Resultados y beneficios..... 57
6.3.7. Lineamientos de despliegue..... 57
7. Testing (Revisión de calidad)..... 58
7.1. Testing de la API..... 58
7.1.1 Estrategia de Testing..... 58
7.1.2. Metodología y Cobertura de Testing..... 59
7.1.3. Métricas y resultados..... 60
7.1.4. Beneficios obtenidos..... 61
8. Post mortem del proyecto..... 61
8.1. Criterios de Éxito..... 61
8.2. Elementos No Contemplados Durante el Desarrollo..... 63
8.3. Consideraciones retrospectivas y oportunidades de mejora..... 65
8.3.2. Tecnologías..... 65
8.3.3. Alcance del backend y frontend..... 66
8.3.4. Uso de bases de datos NoSQL..... 67
8.3.5. Dinámica del equipo..... 68
8.4. Conclusiones y aprendizaje..... 68
9. Memoria del proyecto..... 69
Resumen de proyecto..... 69
Diagrama de Gantt en tiempo real de ejecución..... 70
10. Glosario..... 73
11. Anexos..... 79
Anexo I..... 79
Anexo II..... 80
Anexo III..... 104
Anexo IV..... 133
12. Bibliografía..... 135

1. Agradecimientos

Como equipo, deseamos expresar nuestro más profundo agradecimiento a quienes hicieron posible la exitosa culminación de este Proyecto Final de Grado.

En primer lugar, a nuestros profesores y mentores, por la valiosa guía, la paciencia y la transferencia de conocimientos esenciales para nuestro desarrollo profesional.

Extendemos nuestro reconocimiento a compañeros y amigos por su constante apoyo y compañerismo a lo largo de este trayecto.

Finalmente, un agradecimiento muy especial a nuestras familias, pilar incondicional que nos brindó comprensión, aliento y sostén durante todo el recorrido académico.

2. Resumen del Proyecto

El proceso de aprobación de los Instrumentos A en la Facultad de Ingeniería de la Universidad Nacional de Mar del Plata presenta limitaciones operativas derivadas de su naturaleza manual: tiempos de procesamiento prolongados, ausencia de un sistema centralizado de seguimiento y dependencia de comunicaciones informales entre los actores involucrados.
Para superar estas limitaciones, se diseñó y desarrolló el sistema GIA (Gestión de Instrumentos A), una aplicación web que permite la carga, revisión y validación de Instrumentos A a través de un flujo de trabajo digitalizado. La plataforma implementa un esquema de roles que replica la jerarquía institucional, automatiza las notificaciones entre las distintas etapas de validación y mantiene un registro completo de cada acción realizada sobre los documentos.
El resultado es un sistema funcional que transforma un proceso que anteriormente demandaba hasta dos meses en uno estimado en una semana, eliminando el riesgo de extravío de documentación y brindando visibilidad permanente del estado de cada solicitud.
Este desarrollo representa el primer paso hacia la digitalización de los procesos administrativos del cuerpo docente de la Facultad de Ingeniería, aportando una solución concreta a una necesidad institucional de larga data.

3. Introducción

En la actualidad, dentro de las instituciones académicas existe un documento denominado Instrumento A, el cual funciona como identificador y descriptor de cada asignatura dictada en la facultad. Este documento contiene un desglose completo de la materia, incluyendo la carga horaria, el número de docentes responsables, los contenidos a desarrollar, la bibliografía de referencia, entre otros aspectos relevantes.

La gestión de los Instrumentos A se realiza de manera manual y en formato físico. El proceso comienza con la elaboración del documento por parte del docente a cargo de la asignatura, quien lo entrega al departamento correspondiente. Posteriormente, el director de dicho departamento revisa el contenido y, según corresponda, puede elevarlo a la Secretaría Académica o bien solicitar correcciones al docente responsable. Finalmente, la Secretaría Académica lleva a cabo una última validación que concluye con la aprobación definitiva del Instrumento o con su rechazo, lo que reinicia nuevamente la cadena de revisión.
Este procedimiento presenta múltiples dificultades: cuando existe un gran volumen de Instrumentos A en espera de validación, los tiempos de gestión se incrementan considerablemente. A ello se suman los errores humanos y los problemas de coordinación, lo que genera falencias en la trazabilidad, demoras en la aprobación y una gestión ineficiente del proceso.
Frente a este escenario, surge la necesidad de desarrollar una solución tecnológica que permita optimizar el proceso de gestión, mejorar la trazabilidad y reducir los tiempos de validación de los Instrumentos A, contribuyendo así a la eficiencia y transparencia institucional.

3.1. Objetivos del proyecto

3.1.1. Objetivo general

Desarrollar e implementar una solución tecnológica que permita la digitalización, el control, la trazabilidad y la gestión de los Instrumentos A correspondientes a las asignaturas de la Facultad de Ingeniería de la Universidad Nacional de Mar del Plata.

3.1.2. Objetivos específicos

A lo largo del desarrollo del proyecto se alcanzaron diversos objetivos relevantes, entre los que se destacan:

- Relevar y comprender los procesos actuales de gestión manual de los Instrumentos A, identificando sus principales limitaciones y oportunidades de mejora.
- Digitalizar los Instrumentos A, eliminando la dependencia del soporte papel y centralizando la información en un único sistema accesible.
- Facilitar la carga, revisión y validación de los Instrumentos A por parte de los actores involucrados en el proceso.
- Registrar y hacer visible cada instancia del ciclo de validación, garantizando trazabilidad y transparencia sobre el estado de cada instrumento.
- Reducir los tiempos de gestión mediante la automatización de notificaciones y el seguimiento del flujo de aprobación.
- Sentar las bases para que la solución pueda adoptarse progresivamente en toda la Universidad Nacional de Mar del Plata.

Asimismo, el proyecto representó una oportunidad de aprendizaje y crecimiento profesional para los integrantes del equipo, consolidando competencias aplicables en contextos organizacionales reales.

3.1.3. Alcance y limitaciones

El presente proyecto comprende la digitalización, el control, la trazabilidad y la gestión de los Instrumentos A correspondientes a las asignaturas de la Facultad de Ingeniería. La solución está orientada a docentes, directores de departamento y la Secretaría Académica, brindando herramientas que optimizan el flujo de trabajo y centralizan la información en un único lugar accesible.

La plataforma es accesible desde cualquier dispositivo con conexión a internet, garantizando disponibilidad para todos los actores del proceso sin requerir instalaciones adicionales ni condiciones particulares de hardware.

En cuanto a las limitaciones, el sistema no contempla la integración con otros sistemas institucionales, como la gestión de alumnos o de calificaciones, y su alcance se limita exclusivamente a los procesos vinculados al Instrumento A. Asimismo, la plataforma estará operativa únicamente durante los períodos del calendario académico destinados a la creación de Instrumentos A, permaneciendo inactiva el resto del tiempo.

4. Planificación del proyecto

4.1. Análisis del problema

En la era actual de digitalización, numerosos procesos académicos administrativos continúan ejecutándose de manera manual, generando ineficiencias significativas en las instituciones educativas. La Universidad Nacional de Mar del Plata no es la excepción a esta problemática.

El proceso de aprobación de instrumentos A representa un cuello de botella crítico en la gestión académica institucional. Los instrumentos A son documentos fundamentales que habilitan el dictado de materias durante cada ciclo académico, siendo requisito indispensable su aprobación previa para el funcionamiento normal de la oferta educativa.

El proceso actual presenta las siguientes deficiencias operativas:

Gestión manual fragmentada: La tramitación requiere coordinación constante entre profesores, directores de departamento y secretaría académica, sin un sistema centralizado de seguimiento.

Tiempos de procesamiento excesivos: En el peor de los casos documentados, el proceso de acreditación completo alcanza los dos meses de duración, impactando directamente en la planificación académica.

Falta de trazabilidad: No existe visibilidad del estado actual de cada solicitud, obligando a los actores involucrados a realizar seguimientos manuales constantes.

Riesgo de pérdida de información: Al depender de documentación física y comunicaciones informales, existe riesgo de extravío o demoras no justificadas.

La digitalización del proceso de gestión de instrumentos A surge como una solución estratégica que permitiría:

- Centralizar y sistematizar el flujo de aprobaciones
- Automatizar notificaciones y seguimientos
- Generar trazabilidad completa del proceso
- Reducir significativamente los tiempos de procesamiento

Esta iniciativa fue identificada inicialmente por el departamento de informática institucional y desarrollada colaborativamente con el equipo directivo del proyecto.

4.2. Equipo de trabajo

El desarrollo del proyecto fue llevado a cabo por un equipo multidisciplinario compuesto por cinco integrantes: tres estudiantes de la carrera de Ingeniería en Informática y dos directores de proyecto que proporcionaron supervisión académica y técnica durante todo el ciclo de desarrollo.

Los directores Mg. Spinelli, Adolfo Tomás y Lic. Rico, Carlos Alberto desempeñaron un rol fundamental en la orientación estratégica del proyecto, aportando su experiencia profesional y conocimiento académico a través de reuniones periódicas de seguimiento, revisiones técnicas y retroalimentación constructiva.

El núcleo del desarrollo tecnológico estuvo conformado por los estudiantes **Nahuel Barriga**, **Gregorio Firmani** y **Bruno Trinitario**, quienes establecieron desde el inicio una metodología de trabajo colaborativa basada en la comunicación fluida y la distribución equilibrada de responsabilidades.

Especialización técnica:

- **Nahuel Barriga**: Se especializó en el desarrollo del frontend, siendo responsable de la implementación de las interfaces de usuario (UI), la experiencia de usuario (UX) y la integración con los servicios backend.

- **Gregorio Firmani** y **Bruno Trinitario**: Se enfocaron en el desarrollo del backend, encargándose de la arquitectura del servidor, la lógica de negocio, el manejo de bases de datos y la implementación de APIs.

Es importante destacar que, independientemente de la especialización individual, todas las decisiones críticas del proyecto fueron tomadas de manera consensuada por los tres integrantes del equipo de desarrollo. Esto incluyó:

- Decisiones de diseño de la arquitectura del sistema
- Análisis de requerimientos funcionales y no funcionales
- Diseño de interfaces de usuario y experiencia de usuario
- Selección de tecnologías y herramientas de desarrollo
- Definición de estándares de codificación y metodologías de trabajo

Esta metodología colaborativa aseguró que el proyecto mantuviera coherencia en todos sus aspectos y que cada integrante tuviera una comprensión integral del sistema desarrollado, más allá de su área de especialización técnica.

4.3. Metodologías de trabajo

La selección de una metodología de desarrollo apropiada constituye un factor crítico en el éxito de cualquier proyecto de software, determinando la eficiencia operativa, la calidad del producto final y la capacidad de adaptación ante cambios en los requerimientos. En el contexto de este proyecto, se requería una metodología que permitiera la colaboración efectiva entre un equipo de tres desarrolladores, garantizara la trazabilidad del progreso y facilitara la gestión de riesgos inherentes al desarrollo de sistemas de información.

4.3.1. Metodología Kanban seleccionada

Se optó por la implementación de *Kanban* como *framework* principal de gestión del proyecto. Si bien se consideró el uso de Scrum, se descartó debido a que la rigidez de sus ciclos cerrados (sprints) dificultaba la coordinación con las responsabilidades académicas del equipo y generaba una carga administrativa excesiva para un grupo reducido.
Esta decisión se fundamentó en las siguientes ventajas específicas para el contexto del desarrollo:
Visualización del flujo de trabajo: Kanban proporciona una representación visual clara del estado de todas las actividades del proyecto mediante un tablero organizado en columnas que representan diferentes etapas del proceso de desarrollo: "Backlog", "En análisis", "En desarrollo", "En revisión", "Testing" y "Completadas".
Flexibilidad y adaptabilidad: La metodología permite ajustes dinámicos en las prioridades y alcance del proyecto sin interrumpir el flujo de trabajo establecido, característica fundamental para un proyecto académico con restricciones temporales específicas.

Gestión de trabajo en progreso (WIP): La implementación de límites en el trabajo simultáneo evita la sobrecarga del equipo y mejora la calidad de las entregas.

4.3.2. Estructura operativa implementada

Granularidad de tareas: Se estableció un criterio específico para la definición de actividades, evitando tanto tareas excesivamente amplias (como "desarrollar módulo completo de autenticación") como tareas triviales (como "modificar una línea de código"). El tamaño óptimo de cada tarea se definió entre 2 y 5 horas de desarrollo, permitiendo entregas frecuentes y seguimiento detallado del progreso.
Herramientas tecnológicas: Se implementó app.clickup.com como plataforma digital para la gestión del tablero Kanban, garantizando accesibilidad remota y trazabilidad histórica de todas las actividades del proyecto.

4.3.3. Protocolo de comunicación y seguimiento

La metodología Kanban fue complementada con un protocolo estructurado de comunicación que incluyó:
Reuniones de sincronización semanal: Encuentros presenciales/virtuales de 30 a 90 minutos cada Miércoles, donde cada integrante presenta el progreso realizado, identifica impedimentos y colabora en la resolución de dependencias técnicas entre módulos.
Canales de comunicación asíncrona: Implementación de Whatsapp y Discord para consultas técnicas inmediatas y coordinación diaria, complementando las reuniones formales con comunicación fluida y documentada.

Esta metodología híbrida permitió mantener la agilidad y flexibilidad requeridas para el desarrollo técnico, mientras se cumplían los estándares de rigor y documentación esperados en un proyecto final de carrera de Ingeniería en Informática.

4.4. Análisis FODA

4.4.1. Fortalezas

- **Automatización integral del proceso:** La digitalización completa del flujo de creación, seguimiento y aprobación de Instrumentos A elimina la intervención manual en tareas repetitivas, reduciendo significativamente la probabilidad de errores humanos y optimizando los tiempos de ejecución del proceso administrativo.

- **Arquitectura de roles bien definida:** El sistema implementa una estructura organizacional clara que replica fielmente la jerarquía institucional, incorporando roles específicos para administrador del sistema, docentes solicitantes, docentes observadores, directores de departamento y personal de secretaría académica, garantizando un flujo de trabajo ordenado y responsabilidades bien delimitadas.

- **Apoyo institucional directo:** Al tratarse de un proyecto impulsado por la propia facultad y destinado a resolver una necesidad interna, el equipo de desarrollo contó con acceso directo a los actores involucrados en el proceso de gestión de Instrumentos A. Esta cercanía permitió la supervisión continua del director y el referente funcional del proyecto, lo que facilitó la toma de decisiones informadas y la alineación permanente entre el desarrollo y las necesidades reales de la institución.

4.4.2. Oportunidades

- **Impulso a la transformación digital de la facultad:** El desarrollo del sistema sienta un precedente concreto para la digitalización de otros procesos administrativos de la Facultad de Ingeniería, proporcionando experiencia institucional, infraestructura reutilizable y lecciones aprendidas aplicables a futuras iniciativas.

- **Adopción como estándar institucional:** La exitosa implementación del sistema puede posicionarlo como modelo de referencia para futuras digitalizaciones, incrementando su relevancia estratégica y consolidando su uso como plataforma base para otros desarrollos tecnológicos institucionales.

- **Integración con ecosistemas existentes:** Las tecnologías seleccionadas facilitan la futura integración con sistemas de gestión académica ya implementados en la institución, creando oportunidades para el desarrollo de un ecosistema digital unificado.

4.4.3. Debilidades

- **Curva de aprendizaje tecnológica:** La falta de experiencia previa del equipo de desarrollo en PHP y el framework Laravel representó un desafío inicial que requirió inversión adicional de tiempo en capacitación y familiarización con las herramientas, impactando potencialmente en los tiempos iniciales de desarrollo.

- **Dependencia de recursos humanos especializados:** La operación y mantenimiento continuo del sistema requiere personal con conocimientos técnicos específicos en las tecnologías implementadas, lo que genera una dependencia institucional de perfiles especializados para garantizar la continuidad operativa del sistema.

- **Limitaciones en la gestión de cambios:** La especialización técnica requerida puede generar cuellos de botella para la implementación de modificaciones o mejoras futuras, especialmente si no se cuenta con un equipo de soporte técnico interno adecuadamente capacitado.

4.4.4. Amenazas

- **Resistencia organizacional al cambio:** La transición de procesos manuales a digitales puede encontrar resistencia en usuarios con menor familiaridad tecnológica, afectando la adopción efectiva del sistema.

- **Cambio de autoridades institucionales:** Un eventual cambio en la conducción de la Facultad de Ingeniería o del Departamento de Informática podría derivar en una reconfiguración de prioridades que desestimara o interrumpiera el proyecto, afectando tanto su continuidad durante el desarrollo como su adopción posterior.

- **Riesgos de continuidad operativa:** La concentración en recursos institucionales genera vulnerabilidades relacionadas con mantenimiento, actualizaciones de seguridad y disponibilidad técnica, impactando la estabilidad operacional.

4.5. Solución planteada

La solución propuesta permite la carga, el seguimiento y la aprobación de los Instrumentos A de manera centralizada, integrando a docentes, directores de departamento y Secretaría Académica en un único espacio de trabajo. De esta forma, se elimina la necesidad de intercambios físicos de documentación y seguimientos informales, aportando orden, transparencia y trazabilidad a todo el proceso.
Cabe destacar que el alcance del presente trabajo llega hasta el desarrollo de la solución, quedando fuera del mismo el despliegue en un entorno productivo, la puesta en funcionamiento operativa y los procesos de gestión inicial asociados a su adopción institucional.

4.5.1. Beneficios esperados

La implementación de esta solución digital proporcionará las siguientes mejoras específicas:

- **Reducción significativa de tiempos**: Disminución del período de acreditación de 2 meses a 1 semana aproximadamente
- **Trazabilidad completa**: Seguimiento integral y transparente del ciclo de vida de cada Instrumento A
- **Minimización de errores**: Reducción de errores humanos y eliminación del riesgo de pérdida de información
- **Transparencia en el proceso**: Visibilidad completa del estado y avance de cada solicitud de acreditación
- **Optimización administrativa**: Disminución sustancial de la carga administrativa para todos los actores involucrados

4.5.2. Alcance funcional

El sistema proporcionará las siguientes funcionalidades principales:

- **Gestión integral de Instrumentos A**: Consulta de historial, herramientas de creación facilitada y administración completa del ciclo de vida
- **Sistema de colaboración**: Funcionalidad de comentarios durante las diferentes etapas de validación
- **Notificaciones automatizadas**: Sistema de alertas y recordatorios automáticos para optimizar el flujo de trabajo

4.5.3. Arquitectura tecnológica

Backend - API REST

Para el desarrollo del *backend* se ha seleccionado el *framework Laravel* con *PHP* como lenguaje principal. Esta decisión se fundamenta en:

- **Restricciones institucionales**: Limitación establecida por la Facultad de Ingeniería a las opciones Java o PHP
- **Ventajas de PHP**: Amplia adopción actual en organizaciones, madurez tecnológica, estabilidad comprobada y flexibilidad de desarrollo
- **Ecosistema Laravel**: Framework maduro que permite implementación de estándares modernos y buenas prácticas para el desarrollo de APIs REST robustas, confiables, performantes, seguras y mantenibles

Frontend - Interfaz de usuario

Para la capa de presentación se ha optado por *Next.js* con *TypeScript*, proporcionando:

- **Rendimiento optimizado**: Aprovechamiento del renderizado del lado del servidor (SSR) y generación de sitios estáticos (SSG)

- **Robustez y seguridad**: Tipado estático mediante *TypeScript* para reducir errores en tiempo de ejecución
- **Escalabilidad**: Tecnología moderna y estandarizada que facilita el crecimiento futuro del sistema

Comunicación en tiempo real

La implementación incluye el desarrollo de un módulo de mensajería y notificaciones en tiempo real. Para esto se implementó un servidor de *WebSockets* desarrollado en *Node.js* con *Express*, seleccionados por:

- **Flexibilidad**: Adaptabilidad a diferentes requerimientos de comunicación en tiempo real
- **Compatibilidad**: Integración fluida con el resto del *stack tecnológico*
- **Performance**: Rendimiento optimizado para manejo de eventos y notificaciones instantáneas

4.6. Análisis de riesgos

4.6.1. Proximidad hacia el problema

Descripción: El problema abordado por el proyecto se encuentra dentro de la estructura administrativa de la facultad, un ámbito donde los estudiantes poseen acceso limitado. Esta restricción institucional representa una barrera potencial para la búsqueda de información y la comprensión profunda de los procesos existentes, pudiendo generar retrasos en las actividades de elicitación de requerimientos y análisis del problema.

Impacto potencial: Medio - Podría ocasionar demoras en las fases de relevamiento de información y validación de requerimientos.

Medidas de mitigación:

- Establecimiento de reuniones periódicas estructuradas con el departamento académico para validación continua de requerimientos.
- Documentación exhaustiva de cada sesión de relevamiento.
- Mantenimiento de canales de comunicación formales con los directores del proyecto para resolución de consultas.

4.6.2. Falta de conocimiento técnico

Descripción: Las tecnologías especificadas como requisito institucional representan un desafío significativo dado que el equipo de desarrollo carece de experiencia previa en estas herramientas. Esta brecha de conocimiento técnico constituye un riesgo potencial de generar cuellos de botella durante el desarrollo, afectando tanto los tiempos de implementación como la calidad del código producido.

Impacto potencial: Alto - Podría provocar retrasos sustanciales en los tiempos de desarrollo y la acumulación de deuda técnica debido a implementaciones no óptimas durante la curva de aprendizaje.

Medidas de mitigación:

- Asignación de períodos dedicados específicamente a la investigación y capacitación en las tecnologías requeridas
- Consulta con profesionales experimentados en las tecnologías requeridas para orientación técnica y revisión de código
- Documentación de mejores prácticas identificadas durante el proceso de aprendizaje

4.6.3. Falta de apoyo de terceros

Descripción: La ausencia de recursos externos dedicados al aseguramiento de calidad del software representa un riesgo de sesgo en la evaluación del producto. Sin validación independiente, existe la posibilidad de considerar cumplidos ciertos estándares de calidad debido al cansancio del equipo o por tratarse de los desarrolladores originales del código, comprometiendo la objetividad en la evaluación.

Impacto potencial: Medio/Bajo - Podría resultar en la introducción de defectos no detectados o implementaciones que no cumplan completamente con los estándares de calidad esperados.

Medidas de mitigación:

- Implementación de revisiones de código cruzadas entre miembros del equipo
- Establecimiento de sesiones de validación con los directores del proyecto
- Adopción de prácticas de testing automatizado como mecanismo de validación objetivo
- Documentación de criterios de aceptación claros para cada funcionalidad

4.6.5. Falta de coordinación entre el equipo

Descripción: La diferenciación en el alcance de responsabilidades entre backend y frontend, combinada con variaciones en complejidad técnica y tiempos de resolución, representa un riesgo potencial de desincronización entre los integrantes del equipo. Esta descoordinación podría manifestarse en inconsistencias de código, duplicación de esfuerzos o bloqueos por dependencias no satisfechas.

Impacto potencial: Medio - Podría ocasionar retrasos en los tiempos de desarrollo, generar inconsistencias en la implementación y requerir esfuerzos adicionales de integración.

Medidas de mitigación:

- Implementación de reuniones semanales de sincronización para alineamiento del equipo
- Establecimiento de protocolos de comunicación clara y documentación de decisiones técnicas
- Definición temprana de contratos de API entre frontend y backend para minimizar dependencias

4.7. Estimación de tiempos (Gantt)

La planificación constituye uno de los pilares fundamentales en la gestión de proyectos de desarrollo de software, estableciendo las bases para el éxito del emprendimiento técnico. Esta etapa crítica del ciclo de vida del proyecto define el marco contractual y operativo que regirá la relación entre las partes involucradas: el cliente demandante y el equipo de desarrollo.

A lo largo del proyecto se construyeron dos diagramas de Gantt diferentes. El primero se confeccionó de forma previa a la ejecución técnica, mientras que el segundo se desarrolló como una extensión y reconstrucción del anterior durante las últimas semanas del proyecto; es decir, en la primera quincena del mes de junio

Fase I: Estimación Inicial Pre-Desarrollo

Esta primera etapa de estimación se realizó previo al inicio del desarrollo técnico, esta estimación surge en el dia 0, basándose en:

Análisis de requerimientos funcionales obtenidos durante las sesiones de relevamiento con stakeholders institucionales, permitiendo dimensionar la complejidad del sistema y sus componentes principales.

Evaluación de tecnologías y arquitectura seleccionadas para el proyecto, considerando curvas de aprendizaje del equipo y complejidad de implementación de cada stack tecnológico.

Identificación de dependencias críticas tanto internas (entre módulos del sistema) como externas (integración con procesos institucionales existentes), que podrían impactar los tiempos de desarrollo.

Estimación basada en experiencia previa del equipo en proyectos similares, aplicando técnicas de analogía y juicio experto para calibrar las estimaciones iniciales.

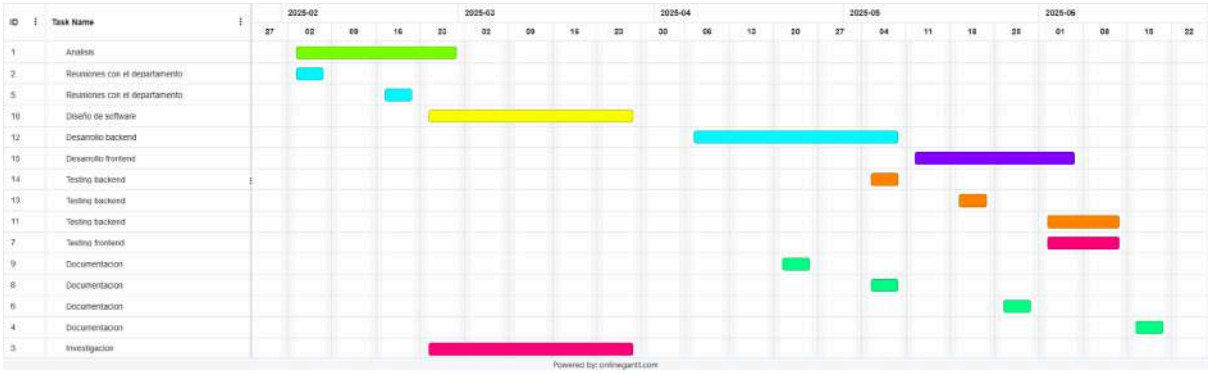


Figura 1. Primer diagrama de Gantt

La Figura 1 presenta el diagrama de Gantt correspondiente a la estimación inicial de la planificación temporal del proyecto, estableciendo la distribución de actividades. El cronograma contempla una carga de trabajo total de 600 horas distribuidas a lo largo del período de desarrollo, lo que equivale a 28.5 horas semanales de dedicación del equipo completo.

Esta planificación se traduce en una dedicación promedio de 5.7 horas diarias para el equipo en su conjunto, representando aproximadamente 2 horas diarias de trabajo efectivo por cada integrante del equipo de desarrollo. La distribución temporal permite mantener un ritmo de trabajo sostenible que se alinea con las responsabilidades académicas de los estudiantes, garantizando la viabilidad de cumplimiento de los objetivos establecidos dentro de los plazos comprometidos.

Fase II: Refinamiento Post Inicio del Desarrollo Técnico

Una vez iniciado el desarrollo técnico, aproximadamente 2 meses luego del comienzo del proyecto, se procedió al refinamiento de las estimaciones originales mediante:

Validación empírica de complejidades identificadas durante la implementación real, permitiendo ajustar estimaciones basándose en datos concretos de productividad y dificultades encontradas.

Recalibración de velocidad del equipo basada en métricas reales de desarrollo obtenidas durante los primeros sprints o iteraciones del proyecto.

Identificación de riesgos emergentes no contemplados en la planificación inicial, que requieren ajustes en cronogramas y asignación de recursos.

Actualización de dependencias y restricciones técnicas descubiertas durante la fase de implementación que impactan la planificación original.

	1 feb	8 feb	15 feb	22 feb	1 mar	8 mar	15 mar	22 mar	29 mar	5 abr	12 abr	19 abr	26 abr	3 may	10 may	17 may	24 may	31 may	7 jun	14 jun
Semanas	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Análisis	7	7.5	7	7.5					2											
Reuniones con el departamento	0.5		0.5																	
Diseño									3	5	5	7	7	5	1					
Desarrollo backend														1	1	5	11	13	11	16
Desarrollo frontend																				16
Testing backend																				
Testing frontend																				
Documentación														1			2		2	
Investigación										2	2				5	8				4
	Durantes estas primeras 3 semanas, resolvimos una parte importante de nuestros requerimientos de software, no solo con reuniones si no tambien plasmando ideas en papel y debatiendo sobre las mejores desiciones. **7.5 Horas/semana**				Mayoritariamente tiempo muerto debido a que debian validar nuestra SRS, y motivos personales de los integrantes. **0 Horas/semana**				Se ultimaron detalles de la fases de analisis y se comenzo el diseño de software **5 Horas/semana**	Continuamos con el diseno del software, al mismo tiempo comenzamos un curso de Laravel para poder familiarizarnos mas rapido con el entorno **7 Horas/semana**				Ultimamos los detalles de los primeros modulos a desarrollar, y al mismo tiempo comenzamos a desarrollarlos, mientras terminabamos con el curso de Laravel **7 Horas/semana**		Se continua con el desarrollo del modulo de usuarios del software, mientras se desarrolla se documento lo desarrollado, esto tambien implico buscar tecnologias y metodos de programacion para contruir un software completo **13 Horas/semana**				Continuamos con el desarrollo del modulo de usuarios, se comienzo con el desarrollo del frontend **36 Horas/semana**
647	7.5	7.5	7.5	7.5	0	0	0	0	5	7	7	7	7	7	7	13	13	13	13	36

21 jun	28 jun	5 jul	12 jul	19 jul	26 jul	2 ago	9 ago	16 ago	23 ago	30 ago	6 sep	13 sep	20 sep	27 sep	4 oct
21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
2		2		1		1		2		1					
				1				1							
2		2		2		1		2		1					
12	14	12	15	12	15	13	12	14	13	13	6				
12	14	12	15	12	15	13	12	14	13	13	6				
6	6	6	2	3	2	3	5	1	4	3	9	2.5	2.5		
		4	2	3	2	3	5	1	4	3	9	2.5	2.5		
2		2		2		2		1		2	4	10	10	10	10
	2		2		2		2		2		2				
Comienzo del modulo de materias y departamentos, junto con el desarrollo de middleware y continuacion de desarrollo de frontend **36 Horas/semana**		Ultimamos el desarrollo del modulo de materias y departamentos, comenzamos el tercer modulo de instrumento A y se continua el desarrollo frontend. Mientras se desarrollan las funcionalidades se testea lo construido **36 Horas/semana**		continuamos el desarrollo del modulo de instrumentos A, se continua el desarrollo frontend. Mientras se desarrollan las funcionalidades se testea lo construido **36 Horas/semana**		Ultimamos el desarrollo del modulo de instrumentos A y se continua el desarrollo frontend. Mientras se desarrollan las funcionalidades se testea lo construido **36 Horas/semana**		Comenzamos el cuarto modulo de mensajeria y notificaciones y se continua el desarrollo frontend. Mientras se desarrollan las funcionalidades se testea lo construido **36 Horas/semana**		Finalizamos el ultimo modulo, al igual que el desarrollo del frontend, se comienzan a realizar test de integracion end to end, y continuamos con la documentacion para la entrega del proyecto **36 Horas/semana**		se finalizan los tests de integracion end to end, y continuamos con la documentacion para la entrega del proyecto **15 Horas/semana**		Se ultiman los detalles del informe y se generan los recursos para la presentación. **10 Horas/semana**	
36	36	36	36	36	36	36	36	36	36	36	36	15	15	10	10

Figura 2. Segundo diagrama de Gantt

La Figura 2 presenta la estimación temporal refinada correspondiente a la segunda fase de planificación del proyecto, reflejando el progreso alcanzado durante las primeras 20 semanas de desarrollo. Esta actualización de la planificación evidencia una recalibración significativa de los cronogramas originales, donde se registra una extensión del proyecto de 16 semanas adicionales respecto a la estimación inicial.

La planificación actualizada incorpora un desglose más granular de las actividades, diferenciando claramente entre los componentes ya desarrollados y aquellos pendientes de implementación, proporcionando mayor trazabilidad y control sobre el avance del proyecto.

5. Ejecución del proyecto

5.1. Análisis del proyecto

La elicitación de requerimientos es una fase crítica que establece los fundamentos técnicos y funcionales del proyecto, definiendo viabilidad y éxito. Esta etapa estructuró el relevamiento de requerimientos mediante una metodología mixta para asegurar la captura integral de necesidades de los *stakeholders* y las restricciones.

La **Metodología de elicitación** incluyó:

- **Supervisión académica especializada** de los directores del proyecto (Mg. Spinelli y Lic. Rico) para priorizar requerimientos.
- **Sesiones de relevamiento estructuradas** semanales (menos de 90 minutos) y sesiones específicas de validación.
- **Proceso iterativo de validación** para alinear el alcance con las expectativas y capacidades técnicas.

Las **Fuentes de requerimientos** identificadas fueron:

- **Dominio del problema:** Flujos de trabajo y criterios de gestión de Instrumentos A.
- **Restricciones institucionales:** Políticas de seguridad, estándares de desarrollo e infraestructura de la UNMdP y Facultad de Ingeniería.
- **Requerimientos regulatorios:** Normativas académicas de acreditación y protección de datos.
- **Requerimientos de escalabilidad:** Consideraciones técnicas para futura expansión a otras unidades académicas.

Las **Técnicas de elicitación** empleadas fueron:

- **Entrevistas semiestructuradas** con *stakeholders* clave.
- **Análisis de procesos existentes** (gestión de Instrumentos A) para identificar puntos de mejora.
- **Revisión documental** de formularios, procedimientos y normativas.

La **Documentación y trazabilidad** se realizó siguiendo estándares de ingeniería de software, incluyendo:

- **Identificación única** (código alfanumérico).
- **Criterios de aceptación** definidos.
- **Prioridad y dependencias** clasificadas.

Esta metodología garantizó la identificación integral de requerimientos, minimizando riesgos y estableciendo bases sólidas para el diseño e implementación.

5.1.2. Modelado del sistema

5.1.2.1. Entidades

El modelado de entidades es esencial en el diseño de sistemas de información. Ofrece una representación abstracta de los elementos del mundo real, facilitando la identificación y estructuración de los datos que el sistema debe gestionar. Esto cimienta el diseño de la base de datos y la arquitectura del software.

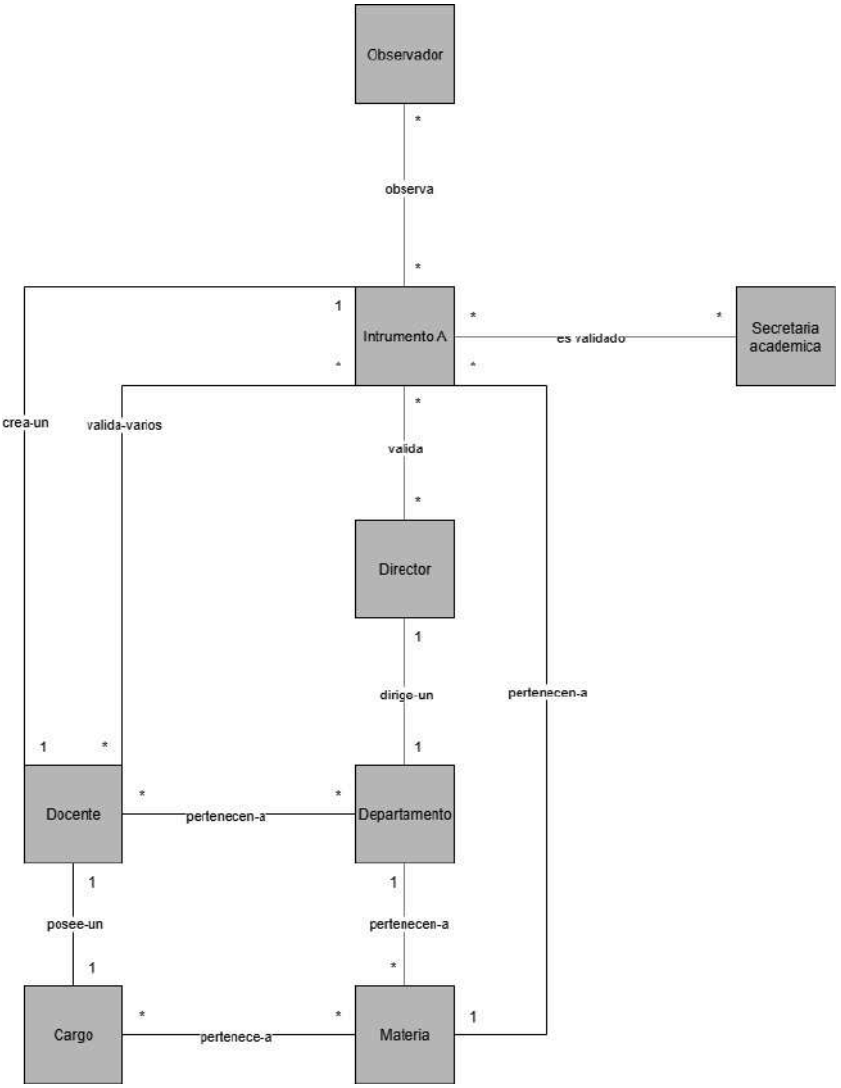


Figura 3. Diagrama de dominio

Entidades principales del sistema

Entidades de roles y usuarios:

- **Usuario:** Entidad base que representa a cualquier persona con acceso al sistema, conteniendo atributos comunes de identificación y autenticación.
- **Administrador:** Entidad especializada responsable de la gestión integral del sistema, configuración de parámetros y administración de usuarios.
- **Director:** Entidad que representa a los directores de departamento, con responsabilidades de supervisión y aprobación de Instrumentos A dentro de su jurisdicción.
- **Docente:** Entidad que modela a los profesores responsables de la creación y mantenimiento de Instrumentos A para las asignaturas a su cargo.
- **Observador:** Entidad que permite el acceso de consulta sin permisos de modificación, facilitando la supervisión y auditoría de procesos.
- **Secretaría Académica:** Entidad representativa del área administrativa responsable de la validación final y aprobación definitiva de Instrumentos A.

Entidades académicas y organizacionales:

- **Departamento:** Entidad que modela las unidades organizacionales de la facultad, estableciendo la estructura jerárquica y de responsabilidades académicas.
- **Materia:** Entidad que representa las asignaturas del plan de estudios, conteniendo información curricular específica y metadatos académicos.
- **Cargo:** Entidad que define las posiciones académicas y sus responsabilidades asociadas, estableciendo la relación entre docentes y materias.

Entidades documentales:

- **Instrumento A:** Entidad central del sistema que modela el documento académico objeto de gestión, incluyendo contenidos programáticos, estados de aprobación, versionado y trazabilidad del proceso de validación.

5.1.2.2. Diagrama de entidad relación

Tras identificar las entidades y sus relaciones conceptuales, se diseña el diagrama entidad-relación (ER). Este plano representa la estructura lógica de los datos, definiendo cómo las entidades se relacionan e interactúan, sirviendo de base para la implementación física de la base de datos.

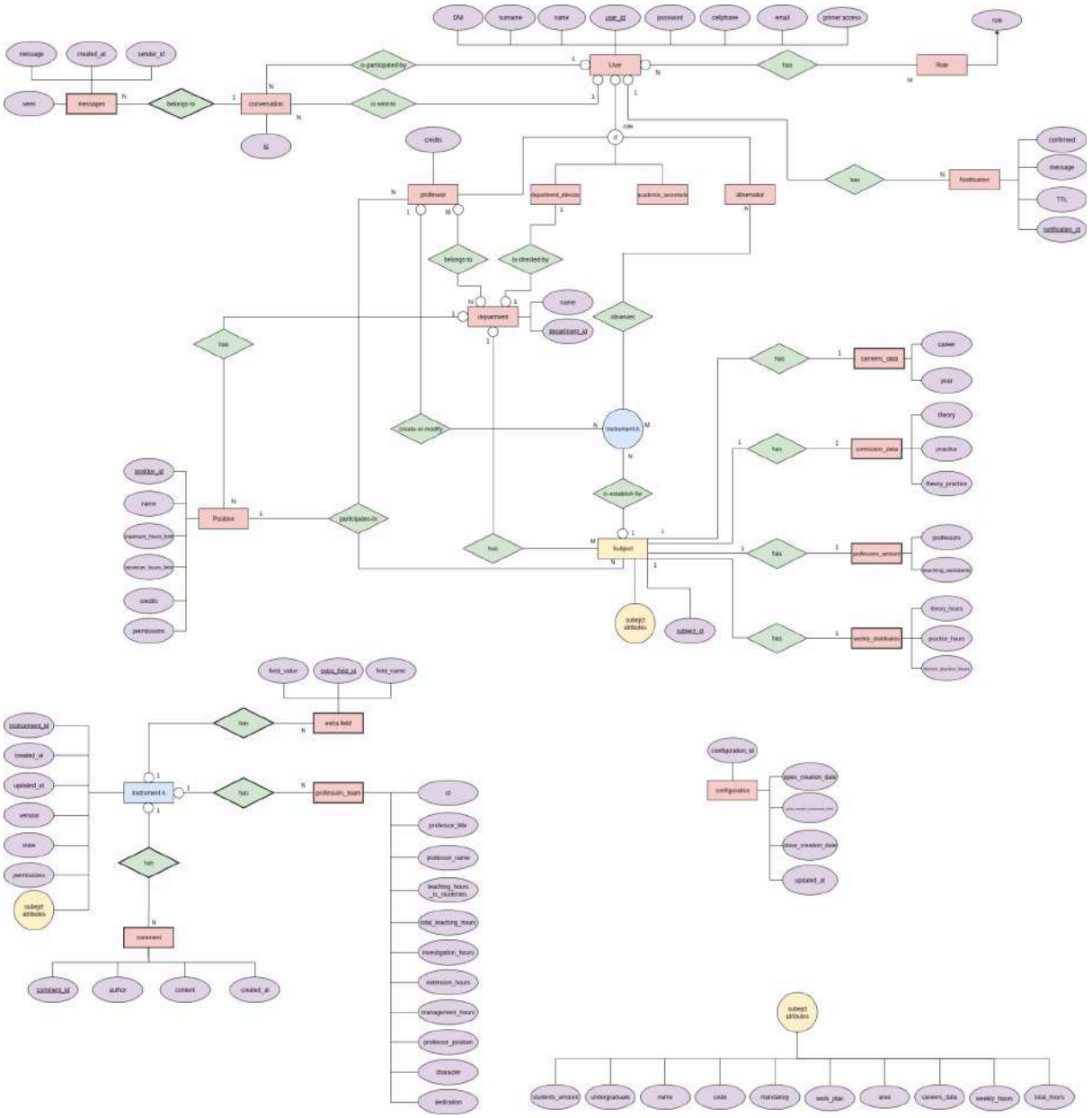


Figura 4. Diagrama de Entidad Relación completo

En la Figura 4 se muestra el diagrama entidad-relación integral del sistema, que ilustra la totalidad de entidades y sus respectivas interrelaciones. El diagrama permite visualizar la estructura modular del sistema y las conexiones entre sus componentes.

Módulo de usuarios y roles

Se establece que un usuario puede tener ninguno, uno o múltiples roles asignados. Asimismo, se incluye una entidad intermedia que gestiona el registro de los roles asociados a cada usuario.

Módulo de mensajería y notificaciones

Se presenta la relación entre las entidades Notificación y Usuario, estableciendo una cardinalidad que permite a un usuario recibir ninguna, una o múltiples notificaciones. Asimismo, los usuarios participan en las conversaciones desempeñando los roles de emisor o receptor. Cada conversación está compuesta por los usuarios participantes y los mensajes intercambiados entre ellos.

Módulo de materias, cargos y departamentos

Se presentan las entidades correspondientes al módulo académico: Director, Departamento, Docente, Cargo y Materia. Se establece que cada director está asignado a un departamento específico, mientras que los docentes pueden pertenecer a ninguno, uno o más departamentos determinados, asimismo también pueden desempeñar diversos cargos en las materias impartidas por dicho departamento, las cuales se conforman de una serie de atributos y relaciones únicas como los datos de la comisión a la que pertenece, la cantidad de docentes y la distribución horaria semanal.

Módulo de instrumento A

Se presenta la extensión del módulo académico mediante la incorporación de la entidad Instrumento A, la cual actúa como elemento integrador del sistema. Se establece que una materia puede generar múltiples instancias de Instrumento A a lo largo del tiempo, manteniendo una relación temporal evolutiva. El Instrumento A hereda los atributos de la entidad Materia e incorpora elementos específicos como equipo docente, campos adicionales y comentarios. Adicionalmente, este módulo introduce dos nuevos tipos de roles: Observador y Secretaria Académica, ampliando las funcionalidades del sistema de gestión.

Módulo de configuración

Se presenta el módulo de configuración del sistema, cuya función principal consiste en establecer las fechas de apertura y cierre para la creación de instrumentos A.

5.1.3. Modelo de casos de uso

El modelo de casos de uso define los **actores** y los **casos de uso**, conceptos clave para la comprensión del sistema.

5.1.3.1. Actores

Representan los roles de los usuarios finales y sus permisos:

- **Observadores**: Externos al proceso, proporcionan retroalimentación objetiva.
- **Docentes**: Núcleo operativo. Crean, modifican y realizan la revisión inicial de los

instrumentos A, iniciando el flujo de validación.
- **Director de Departamento**: Primer nivel de validación formal posterior a la creación por docentes; supervisa el departamento, materias y personal.
- **Secretaría Académica**: Instancia final de validación. Aprueba o devuelve el instrumento A para revisión.
- **Administrador del Sistema**: Actor técnico que configura y mantiene el sistema (creación de departamentos, materias, asignación de roles y permisos).

5.1.3.2. Usos del sistema

Los casos de uso describen las funcionalidades del software, agrupadas en tres categorías:
Casos de Uso Administrativos

Operaciones para configurar y mantener la estructura organizacional:

- Gestión de Materias (CRUD).
- Administración de Departamentos (CRUD).
- Gestión de Cargos.
- Asignación de Roles.

Casos de Uso de Gestión y Análisis de Instrumentos A

Operaciones relacionadas con el ciclo de vida de los instrumentos A:

- Gestión del Instrumento (Creación, modificación, eliminación).
- Sistema de Comentarios.
- Proceso de Validación (flujo de aprobación jerárquico).
- Consulta de Historial (trazabilidad por materia).

Casos de Uso Generales del Sistema

Funcionalidades básicas sin requerir roles específicos:

- Autenticación (Inicio y cierre de sesión).
- Comunicación (Envío/recepción de mensajes).

Esta clasificación estructura las capacidades del sistema y facilita su diseño e implementación.

5.1.3.3. Diagrama de casos de uso

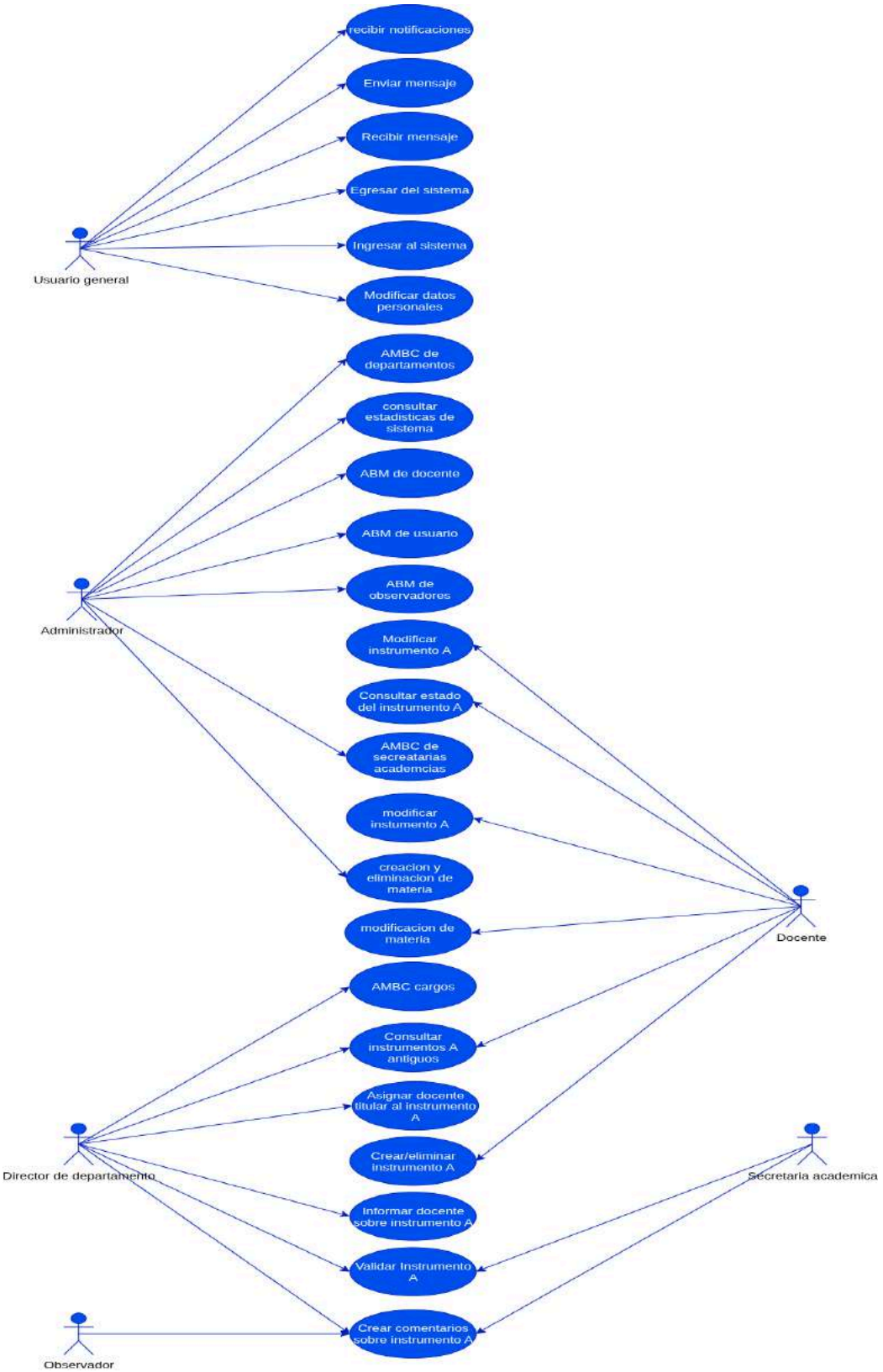


Figura 5. Diagrama de casos de uso

La relevancia de la figura 5 y el Modelo de casos de uso radica en su capacidad para establecer una comunicación clara entre los stakeholders del proyecto, definiendo de manera precisa tanto quién utilizará el sistema como qué funcionalidades específicas necesita cada tipo de usuario para cumplir con sus objetivos organizacionales.

5.1.4. Modelado de procesos del sistema

Esta sección describe el funcionamiento operacional del sistema mediante flujos de trabajo, complementando el análisis estructural con una perspectiva dinámica.

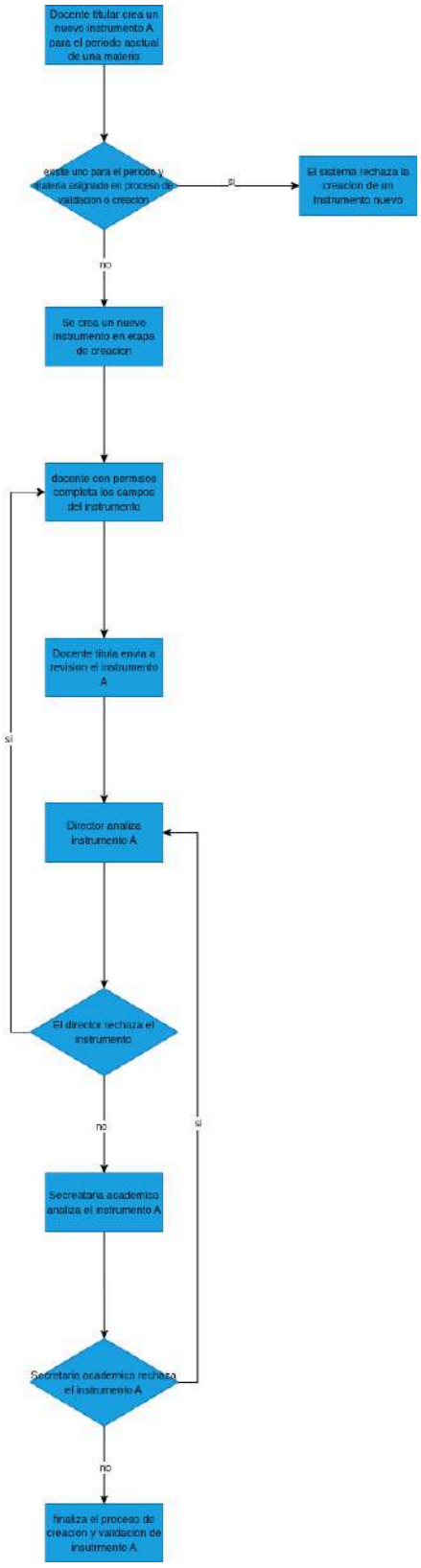


Figura 6 Flujo principal del sistema

Flujo principal: El **Diagrama de Flujo principal (Figura 6)** ilustra el ciclo de vida de los instrumentos tipo A, centrado en su validación, la cual consta de tres etapas secuenciales:

1. **Creación**: Docente responsable.
2. **Validación Departamental**: Director de Departamento.
3. **Validación Institucional**: Secretaría Académica.

El diagrama también incluye puntos de decisión para manejo de aprobación/rechazo y rutas de corrección.Procesos Secundarios

Documentan funcionalidades complementarias, agrupadas en:

- **Gestión Administrativa:** Usuarios/roles, departamentos, materias, cargos, y configuración de permisos.
- **Comunicación, Notificación e ingreso:** Mensajería bidireccional, recuperación de contraseñas, notificaciones automáticas y registro de usuarios.
- **Consulta y Reportes:** Historial de instrumentos A, estadísticas, búsqueda de docentes y exportación a PDF.
- **Gestión Documental:** Versionado tras rechazo, personalización de campos, comentarios/retroalimentación y cálculo automático de créditos docentes.

5.1.5. Análisis de seguridad y permisos

La seguridad utiliza un modelo de **Control de Acceso Basado en Roles (RBAC)** para la integridad y confidencialidad.

5.1.5.1. Sistema de Roles Jerárquicos

Se definen roles jerárquicos que replican la estructura institucional: **Docente** (gestión), **Director de Departamento** (validación departamental), **Observador** (consulta), **Secretaría Académica** (validación final) y **Administrador del Sistema** (privilegios completos). Cada uno tiene acceso específico (ver 5.1.3.2).

5.1.5.2. Mecanismos de Autenticación Segura

Implementa un **protocolo de activación en dos etapas**:

1. **Registro**: El Administrador registra al usuario (correo institucional).
2. **Activación**: El usuario recibe una contraseña temporal cifrada (aleatoria + SHA256) por email, obligatoria de cambiar en el primer acceso.

Las contraseñas se protegen mediante **cifrado irreversible (hash)**.

5.1.5.3. Sistema de Permisos Granulares

Se complementa con **permisos granulares** (manipulación de bits, inspirado en Unix) para mayor flexibilidad.

Permisos de Nivel Instrumento

Los permisos asociados a instrumentos tipo A determinan las **capacidades de interacción específicas** con cada documento:

- **Alcance**: Control sobre la capacidad de generar comentarios y observaciones en instrumentos específicos.
- **Granularidad**: Permite asignación selectiva de permisos de comentario por instrumento individual.
- **Referencia**: Especificaciones detalladas disponibles en Tabla 1.

Bit	Nombre	Descripción
1	Permiso de observador	Controlan si los observadores del sistema pueden crear comentarios sobre el instrumento A
2	Permiso de docente	Controlan si los docente asociados a la materia pueden crear comentarios sobre el instrumento A

Tabla 1. Permisos de Nivel Instrumento

Permisos de Nivel Cargo

Los permisos vinculados a cargos académicos poseen **mayor complejidad y alcance**, abarcando el control integral la materia y sus instrumentos asociados:

Operaciones Controladas:

- Consulta y edición de información de materias
- Creación, modificación y eliminación de instrumentos tipo A
- Gestión de equipos docentes y asignaciones
- Administración de contenidos programáticos

Alcance Departamental: Estos permisos se circunscriben al departamento académico correspondiente, garantizando la segregación de responsabilidades según la estructura organizacional institucional.

Referencia: Matriz completa de permisos disponible en Tabla 1.1

Bit	Nombre	Descripción
1	Permiso de consulta de materia	Controla que los docentes asociados a la materia puedan acceder a la información total de la misma.
2	Permiso de edición de materia	Controla que los docentes asociados a la materia puedan editar información de la misma.
3	Permiso de modificación de Instrumento A	Controla que los docentes asociados a la materia puedan modificar el Instrumento A en proceso de creación actual.
4	Permiso de creación/eliminación de Instrumento A	Controla que los docentes asociados a la materia puedan crear un instrumento A para comenzar con el proceso de validación o eliminar un Instrumento A en proceso de creación.

Tabla 1.1. Permisos de Nivel Cargo

5.1.5.4. Principios de Seguridad Implementados

La arquitectura de seguridad del sistema se fundamenta en los siguientes principios:

- **Principio de Menor Privilegio**: Cada usuario recibe únicamente los permisos mínimos necesarios para cumplir sus funciones.
- **Separación de Responsabilidades**: Las funciones críticas requieren la intervención de múltiples roles para su completitud.
- **Defensa en Profundidad**: Múltiples capas de seguridad (autenticación, autorización, permisos granulares).
- **Trazabilidad**: Registro completo de acciones para auditoría y seguimiento de seguridad.

Madurez Tecnológica

El sistema utiliza tecnologías consolidadas que garantizan estabilidad: **PHP con Laravel** (Backend maduro y soportado), **Next.js con TypeScript** (Frontend moderno, tipado y

optimizado con SSR/CSR) y **MariaDB** (Base de datos empresarial escalable y compatible con SQL).

Mantenibilidad y Sostenibilidad

El diseño es **modular** y facilita el mantenimiento y la evolución. Se aplica **separación de responsabilidades**, **patrones de diseño** reconocidos y **documentación técnica integral** (diagramas, API, despliegue).

Buenas Prácticas de Desarrollo

La implementación sigue estándares: **Versionado de código** distribuido, **testing automatizado** (unitario, integración, funcional) y **configuración basada en entornos** (desarrollo, testing, producción).

Escalabilidad Institucional

La arquitectura soporta el crecimiento institucional:

- **Usuarios concurrentes**: Diseñado para el acceso simultáneo sin degradación, cumpliendo requisitos no funcionales.
- **Distribución de carga**: La arquitectura de APIs REST (Frontend/Backend separados) facilita el balanceo de carga y la distribución geográfica.***Escalabilidad Vertical***
- **Modularidad funcional**: Permite agregar nuevas funcionalidades sin afectar componentes existentes.
- **Flexibilidad organizacional**: Adaptable a diferentes estructuras académicas (otras facultades o instituciones).

5.2. Diseño del sistema

El sistema utiliza una **arquitectura cliente-servidor basada en eventos** con una **base de datos SQL** y un **Servidor de archivos *frontend***. Esta arquitectura centraliza la información para garantizar la gestión, mantenimiento y operación controladas exclusivamente por la institución académica en entornos seguros.

El sistema utiliza una arquitectura **Cliente-Servidor con Eventos**. El servidor principal es una **API REST monolítica modular** (PHP - Laravel) para operaciones síncronas y un servidor **WebSocket** (NodeJS - Express) para comunicación bidireccional y eventos en tiempo real. Un servidor secundario utiliza **NextJS** para el *frontend*, manejando el **SSR** (Server-Side Rendering) y **CSR** (Client-Side Rendering).

La **Base de Datos SQL** se elige por la estructura de información existente, garantizando integridad (transacciones ACID) y permitiendo consultas complejas.

Esta arquitectura ofrece **control institucional completo**, **continuidad operacional** e **independencia de servicios externos**, esenciales para el entorno académico.

La **API REST** estandariza la comunicación síncrona. La arquitectura interna es monolítica modular con un patrón de capas que facilita el desarrollo y el mantenimiento. **WebSockets** permite comunicación asíncrona en tiempo real, mejorando la experiencia de usuario con notificaciones instantáneas. **NextJS** optimiza el rendimiento mediante **SSR** y **CSR**.

La sinergia entre **API REST**, **WebSockets** y **NextJS** logra un rendimiento escalable, flexibilidad arquitectónica y alta mantenibilidad.

5.2.1. Diseño de módulos

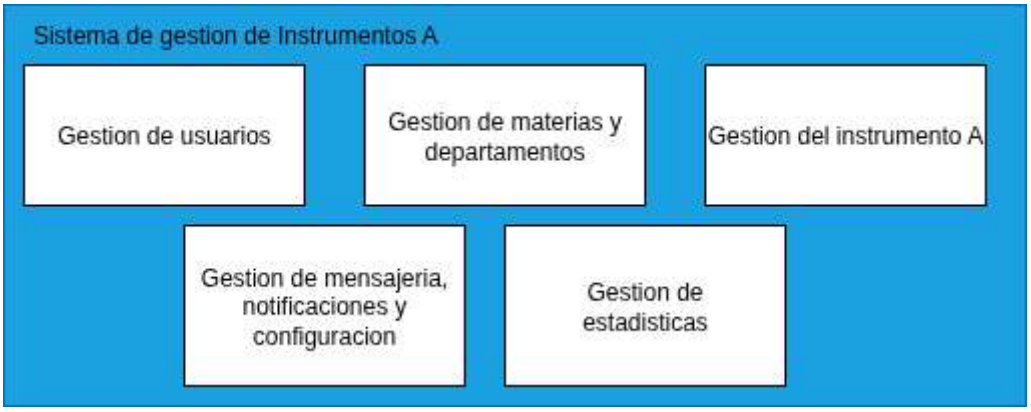


Figura 8. Diagrama de módulos

Como se ilustra en la Figura 8 (Diagrama de módulos), la arquitectura del sistema se estructura en cinco módulos principales: **Usuarios**, **Materias y Departamentos**, **Instrumentos**, **Mensajería, Notificaciones y Configuración**, y **Estadísticas**.

La segmentación del sistema en módulos se basó en los requerimientos funcionales para definir un alcance claro. Esta modularidad facilitó la paralelización, estableció límites definidos y permitió un desarrollo, testing y diseño optimizados.

El documento describe la arquitectura de un sistema modular, centrándose en seis módulos principales:

1. **Gestión de Usuarios:** Núcleo de seguridad y acceso, maneja identidad, autenticación (login, logout, recuperación de contraseña), y administración de perfiles y roles.
2. **Gestión de Materias y Departamentos:** Administra la estructura académica e institucional, incluyendo el CRUD (Crear, Leer, Actualizar, Borrar) de Departamentos, Materias, Cargos (relaciones docente/materia/departamento) y Docentes.

3. **Gestión de Mensajería, Notificaciones y Configuración:** Controla las comunicaciones internas (mensajería), el sistema de notificaciones (incluyendo conectividad WebSocket) y la configuración integral del sistema, gestionando parámetros y caché.
4. **Gestión de Estadísticas:** Módulo analítico, genera consultas de rendimiento y uso (ej. usuarios por rol, Instrumentos A por estado), con acceso restringido al administrador.
5. **Diseño del Módulo de Instrumento A:** Componente central que gestiona el flujo del "Instrumento A" a través de estados (Creación, Validación 1/Director, Validación 2/Secretaría Académica, Finalizado), con funcionalidades como autocompletado, campos extra, comentarios y restricciones específicas (unicidad, períodos, permisos).
6. **Diagrama de Componentes Completo:** Muestra la arquitectura general, diferenciando las interfaces que el sistema consume (dependencias) de las que expone (servicios).

Cada módulo está diseñado con componentes internos desacoplados y expone interfaces bien definidas. El Instrumento A es clave y su diseño detalla el ciclo de vida del proceso de validación por los distintos roles.

5.2.2. Tecnologías

El sistema se desarrolló principalmente con **Laravel** (PHP) como *framework* backend para la API, utilizando exclusivamente sus funcionalidades de servidor. Para la comunicación bidireccional en tiempo real, se incorporó un servidor **WebSocket** independiente.

Tecnologías Backend:

- **Laravel:** *Framework* principal robusto, basado en arquitectura MVC, para el desarrollo ágil y mantenible del backend.
- **Schedule (Cron Jobs):** Biblioteca nativa de Laravel para la programación y ejecución automática de tareas.
- **Bcrypt:** Biblioteca de encriptación que usa *hashing* seguro (ej., SHA-256) para el almacenamiento seguro e inaccesible de contraseñas.
- **MariaDB:** Sistema gestor de bases de datos elegido por su rendimiento, rapidez y escalabilidad.

Tecnologías Frontend:

- **Next.js:** *Framework* de React para construir aplicaciones web de alto rendimiento, optimizadas para SEO mediante *features* como SSR y SSG.

Tecnologías compartidas:

- **JWT (JSON Web Token):** Mecanismo de autenticación *stateless* que permite la identificación de usuarios y encapsula información en cada petición, eliminando sesiones en el servidor. Usado en generación/validación (backend) y almacenamiento/envío (frontend).
- **Node.js y Express.js:** *Runtime* de JavaScript y *framework* minimalista utilizados para crear el servidor WebSocket independiente, facilitando la comunicación asíncrona.
- **Socket.IO:** Biblioteca especializada que implementa la comunicación bidireccional en tiempo real, gestionando usuarios conectados, el envío selectivo de eventos y notificaciones asíncronas entre el servidor WebSocket y el cliente.

5.2.3. Diseño del despliegue del sistema

El despliegue es crucial en el sistema, ya que representa la instancia operativa en producción. Un diseño de distribución adecuado en el servidor es esencial para su comprensión, mantenimiento, y para garantizar rendimiento, escalabilidad y alta disponibilidad.

La estrategia de despliegue se centró en la **contenerización** de servicios (microservicios) para lograr **aislamiento, escalabilidad horizontal, portabilidad** y **gestión simplificada**.

Se eligió **Docker** como tecnología principal por su madurez, soporte y familiaridad, ofreciendo entornos ligeros, reproducibilidad e integración con orquestadores.

El sistema desplegado (Figura 13) consta de:

- **Contenedores de Servicios:** Cada uno incluye la aplicación, *runtime* (Nginx, NextJS, Laravel, Node) y configuración.
- **Red Docker:** Red privada interna (flechas azules) para comunicación segura y eficiente entre contenedores.
- **Proxy Inverso:** Punto de entrada único que gestiona **enrutamiento, balanceo de carga** y **terminación SSL/TLS**.
- **Certificados SSL/TLS:** Aseguran **confidencialidad, integridad** y **autenticación** en la comunicación externa.
- **Segregación de Puertos:** Asignación de un puerto único por servicio dentro de la red Docker para facilitar la **comprensión, depuración** y **documentación**.

6. Desarrollo del proyecto

Este apartado describe el proceso de desarrollo del sistema desde su concepción inicial hasta la integración final de sus componentes. El desarrollo se estructuró en dos áreas principales que fueron ejecutadas de manera paralela por los integrantes del equipo: el **backend** y el **frontend**.

Cada sección detalla las etapas de implementación correspondientes, las decisiones técnicas adoptadas y los desafíos encontrados durante el proceso. Finalmente, se documenta el proceso de integración mediante el cual ambos componentes convergen en un sistema unificado y funcional.

6.1. El Backend

Se define como backend al conjunto de componentes del lado del servidor, incluyendo la lógica de negocio, la gestión de datos, los servicios de red y toda la infraestructura que opera de manera subyacente al sistema. Esta capa es responsable del procesamiento de peticiones, la interacción con bases de datos y la implementación de las reglas de negocio del sistema.

El desarrollo de esta sección del proyecto fue realizado por los estudiantes Firmani, Gregorio y Trinitario, Bruno.

6.1.1. División de trabajo

Un aspecto fundamental en el desarrollo de software es el control de versiones, el desarrollo paralelo y la comunicación asíncrona entre los miembros del equipo. Esta necesidad fue satisfecha mediante el uso de **Git** y **GitHub**, herramientas que facilitaron un desarrollo colaborativo, ordenado e integral del proyecto.

Administración de Módulos y Versiones

Se adoptó una estrategia de administración de código basada en las prácticas utilizadas por grandes organizaciones tecnológicas. El sistema implementado utiliza un manejo de ramas por alcance (*scope*) dentro de cada módulo, siguiendo la nomenclatura estructurada:

[ETAPA]/[MÓDULO]/[ÁREA]/[FEATURE]

Esta convención permite mantener una visión clara y controlada del desarrollo en cada rama, facilitando la trazabilidad y el mantenimiento del código. Los componentes de esta nomenclatura se definen como:

- **ETAPA**: Entorno de trabajo (desarrollo, testing o producción)
- **MÓDULO**: Módulos funcionales del sistema descritos en la sección 5.2.2
- **ÁREA**: Capa de la aplicación (BACKEND o FRONTEND)
- **FEATURE**: Componente específico en desarrollo, siendo las características definidas:
    - MIGRATIONS (migraciones de base de datos)
    - MODELS (modelos de datos)
    - ROUTES (definición de rutas)

- SERVICES (servicios de lógica de negocio)
- CONTROLLERS (controladores)
- REPOSITORIES (repositorios de acceso a datos)
- MIDDLEWARES (middleware de procesamiento)

Esta estructura de ramificación garantiza la organización del código, previene conflictos entre desarrolladores y establece un flujo de trabajo estandarizado para el equipo.

6.1.2. Etapas del desarrollo

Durante el proceso de desarrollo de la aplicación se identificaron dos etapas fundamentales. La primera corresponde al desarrollo planificado, caracterizado por un enfoque iterativo e incremental ejecutado colaborativamente por ambos estudiantes. La segunda etapa se centra en la iteración orientada a la mejora continua del producto, manteniendo la metodología iterativa pero con énfasis en el refinamiento y optimización de la solución.

Primer etapa

Una vez definido el diseño del módulo a desarrollar, incluyendo sus componentes, alcance y funcionalidades, se implementó una estrategia de desarrollo en paralelo. En esta modalidad, Gregorio Firmani asumió la responsabilidad del desarrollo de repositorios y modelos, mientras que Bruno Trinitario se enfocó en la implementación de routers, controladores y servicios. Este enfoque concurrente permitió optimizar los tiempos de desarrollo priorizando la agilidad en la ejecución.

Una vez finalizado el módulo, se procedió a una fase de refinamiento diferenciado: Gregorio Firmani se encargó de la validación mediante la ejecución de pruebas (aspecto que se detalla en secciones posteriores), mientras que Bruno Trinitario asumió el refinamiento, diseño y administración del módulo subsiguiente, garantizando así la continuidad del proceso de desarrollo.

Durante este periodo, el desarrollo de la API, como característica principal del sistema, adoptó un planteamiento **atómico**. Esto implicó una separación clara entre recursos, cada uno asociado a métodos elementales específicos, favoreciendo así la modularidad y el desacoplamiento de las funcionalidades del sistema.

Segunda etapa

Como se mencionó anteriormente, esta etapa adopta un enfoque centrado en la **mejora continua** y el **refinamiento** del software. Una vez completada la implementación de todos los módulos, se destinó un período específico para replantear soluciones e incorporar modificaciones que surgieron durante o posteriormente al desarrollo. Gran parte de estos

cambios fueron concebidos con el objetivo de optimizar la comunicación entre el frontend y el backend, mejorando así la integración y funcionalidad del sistema.

Esta fase fue liderada principalmente por el estudiante Gregorio Firmani, quien se encargó de documentar, implementar y realizar las pruebas correspondientes a las modificaciones mencionadas. Paralelamente, el estudiante Bruno Trinitario inició la elaboración del informe técnico y el proceso de despliegue del sistema en el ambiente de producción.

Durante este periodo, se desarrolló el servidor WebSocket y, paralelamente, se inició un proceso de des-atomización de determinadas secciones de la API que resultaban complejas o excesivamente fragmentadas al momento de establecer la comunicación entre los componentes del sistema. Esta reestructuración tuvo como objetivo simplificar las interacciones y optimizar la eficiencia en el intercambio de datos entre el frontend y el backend.

6.1.3. Soluciones implementadas

Base de datos

Para la gestión de bases de datos, se optó por utilizar únicamente las restricciones nativas ofrecidas por MariaDB, tales como las referencias de claves foráneas, campos no nulos, limitaciones de tamaño en campos de cadenas de texto, entre otras restricciones estándar.

Esta decisión se fundamentó en la necesidad de mantener el control directo desde el backend, lo cual permitió ejercer un manejo más preciso sobre las entidades del sistema. Asimismo, este enfoque proporcionó la flexibilidad necesaria para adaptar la solución en caso de requerir modificaciones durante el desarrollo o mantenimiento del sistema.

Policies

Una característica fundamental del sistema es la seguridad y el control de roles y privilegios, para lo cual se introdujo el concepto de **Policy**. Estos componentes consisten en módulos cuya función principal es actuar como capa de validación entre la solicitud entrante, el controlador y la capa de lógica de negocio.

El uso de Policies se fundamenta en los principios de separación de responsabilidades, alta cohesión y bajo acoplamiento, funcionando como módulos autónomos que no requieren un contexto previo para su ejecución.

Un ejemplo representativo es la validación de permisos docentes sobre una materia determinada. En este caso, el Policy correspondiente verifica, en primer lugar, si el docente posee cargos asociados a la materia en cuestión. De ser afirmativo, procede a validar que el

permiso asignado sea el adecuado para ejecutar la acción solicitada, garantizando así el cumplimiento de las políticas de seguridad del sistema.

Patrón Repository con Clase Base

Para la implementación de los repositorios, se adoptó un enfoque basado en herencia mediante una clase base abstracta (o interfaz genérica), lo cual permitió centralizar métodos comunes tales como métodos de recuperación y eliminación de tablas o de registros específicos a través de id, además de métodos de actualización entre otros.

Esta estrategia de diseño resultó conveniente dado que todos los repositorios del sistema comparten una funcionalidad básica similar, promoviendo así la reutilización de código, la consistencia en las operaciones de acceso a datos y facilitando el mantenimiento del sistema.

Patrón Factory

Este patrón fue utilizado para centralizar la responsabilidad de creación de objetos en el sistema mediante una lógica determinada, promoviendo el principio de responsabilidad única y facilitando la extensibilidad.

Un caso representativo de su implementación es la creación del Instrumento A, el cual puede instanciarse con información previa o sin ella. Dado que este proceso involucra lógica condicional compleja, se implementó un Factory que encapsula y gestiona toda la lógica de creación, determinando el método de instancia apropiado según los parámetros disponibles. Esta solución permite mantener el código cliente desacoplado de los detalles de construcción del objeto y facilitar futuras modificaciones en el proceso de creación.

Patrón DTO

Este patrón fue utilizado para encapsular y estructurar la información transferida entre el backend y el frontend, garantizando que solo se transmitan los datos pertinentes para cada contexto específico. Los DTO actuaron como "moldes" configurables que permiten controlar con precisión qué información se expone en cada operación, evitando la sobre-exposición o sub-exposición de datos.

Un ejemplo representativo se presenta cuando un usuario solicita su información pública. En este escenario, el backend recupera la totalidad de los datos del usuario, incluyendo información tanto pública como privada. Es en este punto donde el patrón DTO interviene, filtrando y estructurando la respuesta para entregar únicamente la información solicitada, protegiendo así los datos sensibles y optimizando la transferencia de información.

Esta implementación no solo refuerza la seguridad del sistema, sino que también mejora el rendimiento al reducir la cantidad de datos transmitidos y facilita el mantenimiento al

desacoplar la estructura interna de las entidades de dominio de las representaciones expuestas externamente.

Patrón Middleware

Este patrón se utiliza como mecanismo de validación de peticiones antes de que estas alcancen las rutas correspondientes. Su funcionalidad principal consiste en rechazar aquellas solicitudes que no cumplen con los estándares de seguridad y autorización definidos por la aplicación.

En el sistema implementado, se estableció una cadena de tres middlewares principales que se ejecutan secuencialmente (cadena de responsabilidades):

1. **Middleware de autenticación JWT**: Verifica la presencia y validez del JWT proporcionado por el sistema, garantizando que la solicitud proviene de un usuario autenticado.
2. **Middleware de autorización por roles**: Valida que el usuario autenticado posea los roles y permisos necesarios para ejecutar la acción solicitada.

Esta arquitectura en capas proporciona un sistema de seguridad robusto y modular, donde cada middleware tiene una responsabilidad específica y claramente definida, facilitando el mantenimiento y la extensibilidad del sistema de control de acceso.

Patron Service Container

El Service Container es un componente inherente al framework Laravel que implementa el patrón de Inyección de Dependencias (Dependency Injection). Este mecanismo gestiona automáticamente la creación y resolución de dependencias entre los diferentes componentes del sistema.

Su funcionamiento se basa en la instanciación dinámica de recursos únicamente cuando son requeridos, evitando así la inicialización anticipada de objetos y previniendo el uso innecesario de memoria. El contenedor mantiene un registro centralizado de las dependencias del sistema y se encarga de inyectar automáticamente en los constructores o métodos que las requieren.

Principios SOLID

Los principios SOLID constituyeron los pilares fundamentales durante el desarrollo del software, proporcionando una metodología robusta y profesional para la construcción del sistema. A continuación, se detalla la aplicación de cada principio:

S - Principio de Responsabilidad Única (Single Responsibility Principle)

Cada clase del sistema fue diseñada para resolver una responsabilidad específica y claramente definida. En los casos donde una clase requiere funcionalidades adicionales, estas se obtienen mediante la implementación de servicios externos, manteniendo así la cohesión y evitando el acoplamiento innecesario.

O - Principio Abierto/Cerrado (Open/Closed Principle)

Las clases fueron diseñadas para estar abiertas a la extensión pero cerradas a la modificación. Esta característica garantiza que, en caso de requerir futuras ampliaciones del sistema o integraciones con otros sistemas, sea posible hacerlo mediante la extensión de clases e interfaces existentes sin necesidad de modificar el código base, preservando así la estabilidad del sistema.

L - Principio de Sustitución de Liskov (Liskov Substitution Principle)

Si bien este principio tuvo una aplicación limitada en el código desarrollado directamente por el equipo debido a la reducida utilización de herencia, es importante destacar que fue ampliamente empleado por las librerías del framework Laravel, garantizando la correcta sustitución de implementaciones en la arquitectura subyacente.

I - Principio de Segregación de Interfaces (Interface Segregation Principle)

Este principio permitió establecer contratos de comportamiento específicos y cohesivos mediante interfaces bien definidas. La segregación adecuada de interfaces contribuyó significativamente a la mantenibilidad y legibilidad del código, evitando que las clases dependan de métodos que no utilizan.

D - Principio de Inversión de Dependencias (Dependency Inversion Principle)

La implementación de este principio fue facilitada por el patrón Service Container y el mecanismo de inyección de dependencias proporcionado por Laravel. Esto permitió que los módulos de alto nivel no dependieran directamente de módulos de bajo nivel, sino de abstracciones, favoreciendo el desacoplamiento y la flexibilidad del sistema.

Express y WebSockets

Como se mencionó anteriormente, Laravel no posee integración nativa con el protocolo de comunicación WebSocket. Ante esta limitación, la solución implementada consistió en el desarrollo de un servidor independiente construido con Node.js.

La arquitectura del servidor WebSocket se compone de los siguientes elementos:

- **Express.js**: Framework utilizado para gestionar la comunicación HTTP entre el servidor WebSocket y la API REST desarrollada en Laravel, permitiendo el intercambio de información y la sincronización de estados.
- **Biblioteca Socket.IO**: Implementación del protocolo WebSocket que permite exponer el servidor para establecer conexiones bidireccionales en tiempo real con el frontend, facilitando la comunicación asíncrona y de baja latencia.

Esta arquitectura establece al servidor WebSocket como un intermediario especializado entre la API REST y el frontend, desacoplando la lógica de comunicación en tiempo real de la lógica de negocio principal del sistema.

Características de la solución:

La implementación se caracteriza por ser:

- **Mantenible**: La separación de responsabilidades entre el servidor WebSocket y la API REST facilita el mantenimiento independiente de cada componente.
- **Ligera**: Node.js proporciona un entorno de ejecución eficiente y de bajo consumo de recursos, ideal para el manejo de múltiples conexiones concurrentes.
- **Robusta**: El manejo nativo de errores asíncronos de JavaScript, combinado con la simplicidad del diseño arquitectónico, garantiza la estabilidad del servidor ante situaciones excepcionales.

Esta solución permitió implementar funcionalidades de comunicación en tiempo real sin comprometer la arquitectura principal del sistema ni requerir cambios significativos en el framework Laravel.

Mensajes y constantes

La implementación de esta solución responde principalmente a dos necesidades: la mantenibilidad del sistema y la potencial extensión a múltiples idiomas en el futuro.

Descripción de la solución:

Se diseñó un sistema de clases especializadas que centralizan todos los mensajes del sistema, categorizados según su naturaleza:

- **Mensajes de éxito**: Confirmaciones de operaciones completadas satisfactoriamente
- **Mensajes de alerta**: Advertencias o notificaciones informativas
- **Mensajes de error**: Indicaciones de fallos o excepciones en las operaciones

Beneficios de la implementación:

Esta arquitectura proporciona múltiples ventajas:

- **Acceso centralizado**: Todos los mensajes del sistema se encuentran en ubicaciones claramente definidas, eliminando la dispersión de literales de texto a lo largo del código.
- **Mantenibilidad mejorada**: Las modificaciones, correcciones o actualizaciones de mensajes se realizan en un único punto, reduciendo el riesgo de inconsistencias y facilitando el mantenimiento.
- **Preparación para internacionalización (i18n)**: La estructura permite una futura extensión a múltiples idiomas de manera eficiente, sin necesidad de refactorizar el código existente. Cada clase puede adaptarse fácilmente para soportar diferentes locales.
- **Consistencia**: Garantiza uniformidad en el tono, formato y estructura de los mensajes presentados al usuario en toda la aplicación.
- **Reutilización**: Los mismos mensajes pueden ser empleados en diferentes contextos sin duplicación de código.

Excepciones

El manejo de excepciones resulta primordial en cualquier sistema de software, ya que garantiza un flujo de ejecución coherente ante situaciones de error y asegura la disponibilidad del sistema frente a fallos críticos o no críticos.

Jerarquía de excepciones personalizadas:

La solución implementada se fundamenta en la creación de excepciones específicas adaptadas al contexto y naturaleza del problema:

ApiException - Excepción base para la API

Dada la naturaleza del backend como API REST, se diseñó una excepción general denominada “ApiException” que encapsula:

- **Mensaje de error**: Descripción detallada del problema ocurrido
- **Código de estado HTTP (status Code)**: Código numérico que indica el tipo de error según los estándares HTTP (4xx para errores del cliente, 5xx para errores del servidor)
- **Stack trace**: Información de depuración sobre la traza de ejecución, disponible cuando es necesario para el diagnóstico

Esta excepción actúa como clase base para otros tipos de errores relacionados con la API, permitiendo un manejo uniforme de las respuestas de error hacia el cliente.

DatabaseException - Excepciones de capa de datos

Considerando la separación de responsabilidades entre sistemas, se implementó la excepción "DatabaseException", especializada en contemplar errores asociados a:

- Fallos de conexión con la base de datos
- Errores en la comunicación con el sistema gestor de base de datos
- Problemas de integridad referencial o restricciones
- Timeouts y problemas de red relacionados con la persistencia

Esta especialización permite al sistema adoptar estrategias específicas de recuperación y disponibilidad ante fallos en la capa de persistencia, tales como:

- Reintentos automáticos de conexión
- Activación de mecanismos de fallback
- Registro detallado de errores para análisis posterior
- Notificaciones a sistemas de monitoreo

Beneficios de la implementación:

- **Granularidad**: Permite manejar diferentes tipos de errores con estrategias específicas
- **Trazabilidad**: Facilita la identificación y diagnóstico de problemas
- **Resiliencia**: Mejora la capacidad del sistema para recuperarse de fallos
- **Comunicación clara**: Proporciona respuestas HTTP apropiadas al cliente según el tipo de error

Cache

Se implementó un sistema de caché estratégico para optimizar el rendimiento del sistema en dos áreas críticas: estadísticas y configuraciones.

Caché de estadísticas:

Las consultas estadísticas representan operaciones computacionalmente costosas, ya que requieren la recuperación y procesamiento de grandes volúmenes de información distribuida en múltiples tablas. Para mitigar este impacto en el rendimiento, se implementó un mecanismo de caché con tiempo de vida (TTL) definido, que almacena los resultados de estas consultas durante un período determinado. Esta estrategia reduce significativamente la carga computacional del servidor al evitar la recalculación constante de las mismas métricas.

Caché de configuraciones del sistema:

Las configuraciones del sistema fueron almacenadas en caché por dos razones fundamentales:

1. **Estabilidad de los datos**: Las configuraciones del sistema presentan una tasa de cambio mínima o nula a lo largo del tiempo, lo que justifica su persistencia en caché sin tiempo de expiración definido.
2. **Alto volumen de consultas**: Múltiples operaciones relacionadas con el Instrumento A requieren acceso frecuente a las configuraciones del sistema. El almacenamiento en caché de estos datos alivia la carga sobre la base de datos y mejora significativamente los tiempos de respuesta.

Selección de tecnología de caché:

Para la implementación del sistema de caché, se evaluaron las siguientes alternativas:

- **Caché en memoria (in-memory cache)**: Descartada debido a que Laravel no proporciona soporte nativo para esta tecnología sin dependencias adicionales.
- **Redis**: Sistema de almacenamiento en memoria de alto rendimiento, considerado como opción viable.
- **Caché basado en base de datos**: Solución seleccionada finalmente.

Justificación de la decisión:

Se optó por implementar el caché basado en base de datos por las siguientes razones:

- **Infraestructura existente**: El servicio de base de datos ya se encontraba desplegado y operativo, eliminando la necesidad de provisionar servicios adicionales.
- **Simplicidad operativa**: Permite aprovechar la infraestructura existente sin agregar complejidad al stack tecnológico.
- **Suficiencia de rendimiento**: Para el volumen y naturaleza de las operaciones del sistema, el caché basado en base de datos proporciona mejoras de rendimiento suficientes sin requerir la complejidad adicional de Redis.
- **Mantenibilidad**: Reduce la cantidad de servicios que deben ser monitoreados y mantenidos en el entorno de producción.

Si bien Redis hubiera proporcionado un rendimiento superior, la decisión tomada representa un balance óptimo entre rendimiento, complejidad y aprovechamiento de recursos existentes.

6.2. El Frontend

Se define como frontend al conjunto de componentes del lado del cliente, incluyendo la interfaz de usuario, la lógica de presentación, la gestión de estado y toda la capa de interacción que opera de manera visible y accesible al usuario. Esta capa es responsable de la renderización de vistas, la captura de entradas del usuario, la validación en tiempo real y la comunicación con el backend a través de servicios de red.

6.2.1. División de trabajo

El control de versiones y la organización del código fueron aspectos fundamentales en el desarrollo del frontend. Esta necesidad fue satisfecha mediante el uso de Git y GitHub, herramientas que facilitaron un desarrollo ordenado e integral del proyecto.

El desarrollo de esta sección del proyecto fue realizado por el estudiante Barriga, Nahuel.

Administración de Módulos y Versiones

Para la estructuración del código se implementaron diversas estrategias, teniendo como prioridad la reusabilidad y la mantenibilidad del sistema. La organización siguió los siguientes principios:

Estructura de ruteo mediante Next.js App Router: Se aprovechó el sistema de enrutamiento basado en carpetas que proporciona Next.js para estructurar los archivos y la lógica de ruteo de forma intuitiva y fácilmente navegable. Esta convención permite mantener una correspondencia directa entre la estructura de directorios y las rutas de la aplicación.

División por roles: Se implementó una segregación de páginas según los diferentes roles de usuario del sistema, permitiendo manejar la lógica de cada perfil de manera aislada. Estas páginas invocaban componentes reutilizables con el objetivo de mantener el código repetido al mínimo y maximizar la reutilización.

Módulos utilitarios (Helpers): Se implementaron archivos especializados conocidos como “helpers", donde se manejó la lógica correspondiente a las estructuras de datos en el frontend, promoviendo la separación de responsabilidades y facilitando el mantenimiento.

Capa de comunicación centralizada: Se desarrolló un componente que centralizó todas las llamadas y respuestas hacia la API, permitiendo la implementación de interceptores para el manejo uniforme de peticiones y respuestas. Este componente operó del lado del servidor, pudiendo gestionar de forma segura e invisible al usuario los tokens de autenticación y los datos sensibles.

Arquitectura de Single Page Application (SPA): El sistema fue desarrollado como una aplicación de página única, proporcionando una experiencia de usuario fluida mediante la navegación sin recarga completa de página y la actualización dinámica del contenido.

Esta estructura de organización garantizó la mantenibilidad del código, facilitó el desarrollo colaborativo y estableció un flujo de trabajo estandarizado para futuras extensiones del sistema.

6.2.2. Etapas del diseño

El desarrollo del frontend se estructuró en tres etapas fundamentales: dos etapas de diseño previas al desarrollo del código y una etapa de implementación. Este enfoque metodológico permitió una planificación exhaustiva antes de la codificación, reduciendo así los riesgos de refactorización y garantizando la coherencia arquitectónica del sistema.

Primera etapa de diseño

En esta fase inicial se desglosaron los requerimientos funcionales (RF) y se categorizaron por rol y por página dentro del software. El objetivo principal fue garantizar el cumplimiento pleno de los requisitos funcionales, asegurar una experiencia de usuario óptima y generar una guía clara para el desarrollo posterior.

Se utilizó Figma como herramienta de diseño para plantear mockups no funcionales del software, permitiendo visualizar la interfaz antes de su implementación y facilitar la retroalimentación temprana sobre aspectos de usabilidad y diseño visual.

Durante esta etapa también se definieron:

- **DTOs y tipos de datos**: Estructuras de transferencia de datos y definiciones de tipos a ser utilizados por los diferentes componentes y roles de usuario.
- **Esquemas de datos:** Se planteó esquemas de datos y límites de los mismos para futura utilización mediante Zod.
- **Módulos utilitarios**: Planificación del desarrollo de utilidades para evitar el acoplamiento de componentes y mantener la lógica modular.
- **Modelos de mocks de datos para simular API:** Planificó y diseñó funcionalidades de la API utilizando MSW para poder interceptar requests y simular respuestas.

Segunda etapa de diseño

Esta etapa se centró en la optimización del rendimiento y la seguridad mediante la categorización estratégica de componentes. Se determinó qué componentes serían renderizados en el navegador del cliente (Client-Side Rendering - CSR) y cuáles en el servidor del frontend (Server-Side Rendering - SSR).

Next.js ofrece la posibilidad de utilizar SSR para aquellos componentes que manejan lógica sensible que debe permanecer transparente a los usuarios, tales como:

- Gestión de tokens de acceso
- Configuración del sistema
- Información sensible del usuario

Además del beneficio en seguridad, el SSR proporciona ventajas significativas en rendimiento, especialmente en equipos con menores capacidades de procesamiento. Con el

objetivo de maximizar la performance, también se implementó pre-renderizado en las diferentes vistas del sistema.

Durante esta fase se evaluaron técnicas adicionales de optimización como “skeletons” y “fallbacks”, las cuales fueron finalmente descartadas debido a que el equipo consideró que el esfuerzo adicional de implementación no se justificaba para el alcance definido del sistema.

6.2.3. Etapa de desarrollo

En esta fase, todo lo planteado en las etapas de diseño fue traducido a código. Se implementó un desarrollo por módulos, manteniendo coherencia con la estrategia utilizada en el desarrollo de la API.

La estrategia principal consistió en priorizar la funcionalidad inicial sobre los aspectos visuales, para posteriormente refinar los componentes de presentación. Este enfoque se materializó mediante la generación de estructuras básicas suficientes para visualizar datos y simular el flujo de información a través del sistema.

Para coordinar el desarrollo lógico-funcional, se utilizó un documento compartido con el equipo de backend que detalla:

- Rutas de la API
- Estructura de datos a enviar en las peticiones
- Formato de datos esperados en las respuestas
- Códigos de error y éxito

Este documento fue objeto de iteraciones y modificaciones constantes a lo largo del desarrollo con el objetivo de optimizar la comunicación entre ambas capas del sistema.

Es importante destacar que se generaron múltiples iteraciones sobre las etapas de diseño durante este período, buscando una mayor refinación en la estructura del código y en la división estratégica del renderizado.

6.2.4. Soluciones implementadas

Mock Service Worker (MSW)

El desarrollo del frontend fue llevado a cabo de forma asincrónica y aislada con respecto al desarrollo del backend. Consecuentemente, se requirió una solución para simular una conexión a la API lo más fiel posible a la realidad sin depender de la disponibilidad del backend. Esta librería permite no solo simular datos y códigos de respuestas, sino que también permite simular diferentes tiempos de las mismas y errores.

Se seleccionó Mock Service Worker (MSW) como herramienta de simulación. Esta librería ofrece módulos que interceptan las peticiones HTTP enviadas al backend y permiten describir el comportamiento esperado del servidor.

Zod

Se utilizó Zod como librería de validación y esquematización de datos, particularmente útil dado que la mayor parte del sistema está compuesta por formularios de entrada de información.

La integración de Zod permitió establecer una capa robusta de validación del lado del cliente, reduciendo peticiones inválidas al backend y mejorando la experiencia de usuario mediante retroalimentación inmediata.

Gestión de Cookies

Para la gestión de sesiones y almacenamiento de tokens de autenticación, se implementó un sistema de cookies del lado del servidor con las siguientes características de seguridad:

- HTTP-Only Cookies: Los tokens de autenticación se almacenan en cookies con la bandera httpOnly, impidiendo que el código JavaScript del cliente acceda a ellos. Esta medida protege contra ataques XSS.
- Secure Flag: Las cookies se configuran con la bandera secure, garantizando que solo se transmiten a través de conexiones HTTPS cifradas, protegiendo contra ataques de tipo man-in-the-middle.
- Same Site Policy: Se implementó la política SameSite para prevenir ataques CSRF (Cross-Site Request Forgery), restringiendo el envío de cookies a peticiones originadas desde el mismo sitio.

Esta estrategia de gestión de sesiones proporciona una capa robusta de seguridad sin comprometer la experiencia de usuario, manteniendo las sesiones activas de forma transparente.

Patrón DTO (Data Transfer Object)

Similar a la implementación en el backend, se utilizó el patrón DTO para estructurar y validar la información transferida entre el frontend y el backend. Los DTO actuaron como contratos de datos que garantizan la integridad de la información en ambas direcciones de comunicación.

Patrón Middleware

Middleware de autenticación: Verifica que el usuario posea una sesión válida antes de permitir el acceso a rutas protegidas. En caso de detectar una sesión inválida o inexistente, redirige automáticamente al usuario a la página de inicio de sesión.

Middleware de autorización por roles: Evalúa los permisos del usuario autenticado para determinar si posee acceso a determinadas secciones o funcionalidades del sistema. Este middleware previene el acceso no autorizado a recursos mediante redirección o denegación de acceso según corresponda.

Middleware de navegación: Gestiona aspectos de la experiencia de usuario durante la navegación, como la persistencia de estado de scroll o la precarga de datos para rutas frecuentemente visitadas.

Esta arquitectura en capas de middleware proporciona un mecanismo flexible y extensible para implementar lógica que debe ejecutarse antes de renderizar componentes, facilitando la separación de responsabilidades y mejorando la mantenibilidad del sistema.

WebSockets

Para implementar funcionalidades de comunicación en tiempo real, se integró un cliente WebSocket en el frontend que establece conexión con el servidor WebSocket.

La implementación incluye:

Gestión de conexión: Establecimiento automático de conexión al iniciar la aplicación y reconexión automática ante desconexiones transitorias.

Manejo de eventos: Sistema de suscripción a eventos específicos que permite a los componentes reaccionar a notificaciones del servidor en tiempo real.

Integración con estado de React: Los mensajes recibidos vía WebSocket se integran con el sistema de gestión de estado de React, actualizando la interfaz de forma reactiva cuando se reciben nuevas notificaciones.

Manejo de errores: Implementación de estrategias de recuperación ante fallos de conexión, incluyendo reintentos con backoff exponencial.

Server-Side Rendering (SSR) y Client-Side Rendering (CSR)

La arquitectura del frontend aprovecha las capacidades de renderizado híbrido que proporciona Next.js, utilizando estratégicamente SSR y CSR según los requisitos de cada componente.

Server-Side Rendering (SSR):

Se utilizó SSR para componentes que:

- Manejan información sensible o credenciales: Carga de datos de configuración, mensajes y notificaciones.
- Requieren datos que deben estar disponibles en la carga inicial: Componentes estáticos como headers o la barra de navegación.
- Componentes no dinámicos: Formularios.

Client-Side Rendering (CSR):

Se utilizó CSR para componentes que:

- Requieren interactividad compleja y manejan estado local frecuentemente actualizado: Todos los componentes de tablas.
- Dependen de eventos del navegador o APIs del cliente: Panel de estadística.

Esta estrategia de renderizado híbrido optimiza tanto el rendimiento como la seguridad, aprovechando las fortalezas de cada enfoque según el contexto específico de cada componente del sistema.

6.3. Integración

6.3.1. Descripción general

Como fase final del desarrollo, se establece que ambos proyectos (backend y frontend) deben converger en un sistema unificado. Este proceso implica verificar que las interfaces de comunicación se respeten y que el comportamiento del software sea el esperado conforme a las especificaciones establecidas previamente.

La fase de integración abarcó dos componentes fundamentales: el aseguramiento de la calidad del software mediante pruebas de extremo a extremo (end-to-end), y la implementación de modificaciones necesarias para posibilitar la integración efectiva de ambos módulos del sistema.

6.3.2. Estrategia de integración

Se adoptó un enfoque basado en control de versiones mediante la creación de una rama específica denominada "integration" en el repositorio de Git. Esta rama tiene como responsabilidad centralizar los archivos de ambos proyectos simultáneamente y recibir los cambios que faciliten el proceso de integración.

Containerización mediante Docker

Cada componente del proyecto fue dockerizado con los siguientes objetivos:

- Lograr un mayor control sobre la ejecución del sistema
- Garantizar la reproducibilidad del entorno de desarrollo y pruebas
- Facilitar el despliegue en diferentes ambientes
- Aislar dependencias y evitar conflictos entre versiones
- Simplificar la configuración del entorno de pruebas

La arquitectura Docker implementada incluye:

- Contenedor para el servidor backend
- Contenedor para el servidor WebSocket
- Contenedor para la aplicación frontend
- Contenedor para la base de datos
- Red Docker personalizada para la comunicación entre servicios

6.3.3. Proceso de integración

El flujo de integración implementado siguió un proceso iterativo basado en las siguientes etapas:

1. **Definición de flujos de uso**: Se especificaron casos de uso críticos que abarcan las funcionalidades principales del sistema, priorizando aquellos que involucran la interacción completa entre frontend y backend.
2. **Ejecución de pruebas**: Se ejecutaron los flujos de uso determinados en el entorno dockerizado, simulando condiciones de operación reales.
3. **Análisis y corrección de errores**: En caso de detectar errores o comportamientos inesperados, se procedió con:
    - Registro detallado del error en el sistema de seguimiento
    - Análisis de causa raíz
    - Implementación de correcciones en la rama correspondiente
    - Merge de cambios a la rama de integración
    - Re-ejecución de las pruebas
4. **Validación y cierre**: En ausencia de errores, se consideró el flujo como validado y finalizado, documentando los resultados obtenidos.

6.3.4. Pruebas end-to-end

Las pruebas de extremo a extremo se enfocaron en validar:

- Correcta comunicación entre frontend y backend a través de las APIs REST definidas
- Consistencia de datos entre las capas del sistema
- Manejo adecuado de estados y errores

- Rendimiento del sistema bajo condiciones normales de operación
- Compatibilidad de las interfaces de usuario con los endpoints del backend

6.3.5. Distribución de responsabilidades

La organización del equipo durante esta etapa se estructuró de la siguiente manera:

Frontend (Nahuel):

- Implementación de modificaciones en componentes React
- Ajustes en las llamadas a la API
- Corrección de validaciones y manejo de respuestas
- Mejoras en la interfaz de usuario basadas en los resultados de las pruebas

Backend y DevOps (Gregorio y Bruno):

- Ejecución de pruebas de los distintos flujos de uso
- Realización de ajustes en los endpoints y servicios del backend
- Administración de la infraestructura Docker
- Configuración de la red y comunicación entre contenedores
- Monitoreo de logs y métricas del sistema

6.3.6. Resultados y beneficios

Esta distribución de tareas permitió:

- **Optimización de tiempos**: Reducción significativa en los ciclos de prueba y desarrollo mediante trabajo paralelo
- **Mejora continua**: Establecimiento de un proceso iterativo que permite identificar y resolver problemas de manera ágil
- **Documentación de despliegue**: Definición de lineamientos y procedimientos aplicables para la implementación del software en un entorno de producción real
- **Estándares de calidad**: Establecimiento de criterios de aceptación claros para cada flujo de uso
- **Trazabilidad**: Registro detallado de problemas encontrados y soluciones implementadas

6.3.7. Lineamientos de despliegue

Como resultado del proceso de integración, se establecieron las siguientes pautas para un eventual despliegue en producción:

- Utilización de Docker Compose para orquestar los contenedores

- Definición de variables de entorno para configuración específica del ambiente
- Implementación de estrategias de respaldo y recuperación de datos
- Establecimiento de procedimientos de monitoreo y logging
- Documentación de dependencias y requisitos del sistema
- Protocolos de actualización y rollback en caso de fallos

7. Testing (Revisión de calidad)

7.1. Testing de la API

El proceso de testing de la API se consideró una parte fundamental del proyecto, al igual que el análisis y desarrollo, con el objetivo de asegurar la calidad de nuestro producto en un ambiente controlado y seguro, proporcionando confianza sobre la estabilidad del sistema antes de su puesta en producción.

7.1.1 Estrategia de Testing

Marco conceptual

La estrategia de testing de la API se fundamentó en el framework **Pest PHP**, seleccionado por su capacidad de generar tests comprensibles y mantenibles. Se optó por esta herramienta sobre alternativas como **Postman** o **Thunder Client**, las cuales se enfocan en el testing manual e interactivo de APIs, debido a dos razones principales:

1. Una mayor integración con el entorno de desarrollo PHP.
2. Proporciona un registro permanente de los tests realizados dentro del código del proyecto, complementando la documentación independiente desarrollada paralelamente.

Filosofía

Se adoptó un enfoque de testing que privilegia la **cobertura funcional exhaustiva** sobre la cobertura de código pura, asegurando que todos los flujos críticos de negocio estén validados bajo diferentes escenarios y roles de usuario. Esta filosofía se sustenta en tres principios fundamentales:

- **Validación por roles:** Cada funcionalidad del sistema fue testeada para todos los roles con acceso a ella, garantizando que las restricciones de autorización funcionen correctamente y que cada tipo de usuario solo pueda realizar las operaciones permitidas según sus privilegios.
- **Testing de reglas de negocio:** Se priorizó la validación exhaustiva de las reglas específicas del dominio académico, como la gestión de créditos de profesores, las transiciones de estado de los Instrumentos A y las validaciones de integridad referencial entre entidades.

- **Cobertura de casos límite:** Además de los flujos normales, se implementaron tests específicos para casos extremos, errores esperados y situaciones de excepción que podrían comprometer la estabilidad del sistema.

7.1.2. Metodología y Cobertura de Testing

Configuración del entorno

Para garantizar la confiabilidad y reproducibilidad de los tests, se implementó un entorno de testing completamente aislado mediante las siguientes estrategias:

- **Aislamiento de la base de datos:** Cada test se ejecuta sobre una base de datos limpia mediante el trait **RefreshDatabase** de Laravel, eliminando dependencia entre tests y asegurando resultados consistentes.
- **Factories para datos de prueba:** Se desarrollaron factories para todos los modelos del sistema, permitiendo la generación consistente y controlada de datos de prueba, incluyendo relaciones complejas entre entidades.
- **Utilities de debugging:** Se implementó herramientas de debugging que facilitan el desarrollo y mantenimiento de tests sin afectar el rendimiento en producción.

Unit tests

El objetivo de los unit tests fue validar de manera aislada los componentes individuales del sistema. De esta manera se garantizaron dos aspectos fundamentales: el funcionamiento correcto individual de cada componente, y la aislación de errores en los feature tests, al contar con la certeza de que ciertos componentes ya habían sido testeados y funcionaban correctamente.

Para este proceso, en primera instancia se validaron los repositorios, asegurando que la recuperación de información de la base de datos se realizará correctamente. Para garantizar un uso más eficiente del tiempo de desarrollo, no se realizaron tests exhaustivos de todos los repositorios, sino que se priorizaron casos genéricos representativos y métodos con lógica específica de mayor complejidad.

Posteriormente, se procedió al testing de los servicios, donde se validó la mayor parte de los métodos implementados. Esta etapa sirvió además como una segunda verificación del funcionamiento correcto de los repositorios, permitiendo validar indirectamente incluso aquellos métodos de repositorios que no habían sido testeados de manera aislada en la etapa previa.

La validación de los servicios también cumplió el objetivo de poner a prueba las reglas de negocio del sistema, validando aspectos críticos como la gestión de los créditos de

profesores, manejo de estados y versiones del instrumento A, y la integridad referencial del sistema.

Feature tests

A partir de los feature tests, se validaron funcionalidades completas del sistema a través de peticiones HTTP, simulando el comportamiento de usuarios finales. A diferencia de los repositorios y servicios, en este caso fueron testeados todos los endpoints de la API, para así poder poner a prueba distintos aspectos del sistema, como lo son:

- **Flujos del sistema:** Al validar todos los endpoints de la API, se verificó el correcto funcionamiento de los flujos del sistema, lo que permitió asegurar la devolución correcta de los códigos de respuesta HTTP y las estructuras de datos JSON. Esto también permitió captar errores que no fueron detectados en los unit tests.

- **Seguridad y autenticación:** A partir de los tests, se pudieron validar los distintos mecanismos de seguridad que se implementan en el sistema, como lo son:
  - El uso de **JWT** para el acceso a la API.
  - El **sistema de autorización por roles**, en el cual se probó para cada endpoint que todos los usuarios con acceso al mismo recibieran una respuesta acorde a su rol.
  - El **sistema de permisos granulares**, para el cual se validaron en los flujos correspondientes, que los profesores tuvieran o no, un cargo con los permisos necesarios para realizar determinadas acciones.

- **Servicios externos:** A través del uso de mocks, se pudo verificar la correcta integración del sistema de notificaciones y envío de emails, sin necesidad de depender de los servicios reales.

7.1.3. Métricas y resultados

Categoría	Archivos	Tests realizados
Feature Tests	12	75+
Unit Tests - Servicios	11	70+
Unit Tests - Repositorios	4	20+
Total	**27**	**165+**

Tabla 2. Cobertura general de tests

7.1.4. Beneficios obtenidos

El proceso de testing llevado a cabo proporcionó distintos beneficios al proyecto:

- **Detección temprana de errores:** Al adoptar un enfoque de testing progresivo, testeando el sistema cada vez que un módulo era finalizado, se logró detectar varios errores y puntos de falla que podrían haber generado una deuda técnica muy costosa en el largo plazo.
- **Refactoring seguro:** La cobertura exhaustiva proporcionó una mayor comprensión del código, lo que brindó la confianza necesaria para realizar cambios mayores dentro del sistema sin temores de introducir retrasos significativos en los tiempos de trabajo.
- **Documentación viva:** Los tests ofrecen un nivel adicional de documentación del sistema, facilitando el trabajo a futuros desarrolladores y proporcionando ejemplos concretos del funcionamiento del sistema.
- **Confianza en el sistema:** Testear la API de manera aislada con herramientas que aseguran su funcionamiento proporcionó mayor confianza a la hora de integrar la API con el resto de los componentes del sistema.

8. Post mortem del proyecto

Esta sección está dedicada a la reflexión crítica sobre tres aspectos fundamentales del proyecto:

En primer lugar, analizaremos las expectativas sobre el futuro del sistema una vez finalizado el desarrollo, considerando su evolución, mantenimiento y posibles escenarios de uso en producción.

En segundo lugar, identificaremos aquellos elementos que no fueron contemplados durante el desarrollo y que, en retrospectiva, hubiesen merecido mayor atención o una consideración más profunda en las etapas de planificación e implementación.

En tercer lugar, exploraremos las funcionalidades y mejoras que nos hubiera gustado implementar de haber contado con mayor disponibilidad de tiempo y recursos, o en ausencia de las restricciones técnicas del sistema actual.

Finalmente, concluiremos con las lecciones aprendidas durante el proyecto y nuestro punto de vista sobre cómo se llevó a cabo el desarrollo en su conjunto.

Este ejercicio de análisis retrospectivo busca no solo documentar las lecciones aprendidas, sino también establecer una base de conocimiento valiosa para futuros proyectos similares.

8.1. Criterios de Éxito

Los criterios de éxito establecidos para el proyecto se fundamentan en tres pilares fundamentales: el uso integral del sistema como plataforma centralizada de gestión, la reducción significativa de los tiempos de validación de Instrumentos A, y el establecimiento de un flujo trazable y ordenado que garantice la transparencia y el control de estos procesos burocráticos.

8.2.1 Estrategia de Adopción Progresiva

La estrategia de implementación se diseñó siguiendo un modelo de adopción gradual que minimiza riesgos y maximiza el aprendizaje institucional. La primera fase contempla que el Departamento de Informática actúe como usuario pionero del sistema, siendo los primeros en tener acceso al sistema operativo. Esta etapa piloto permitirá generar feedback inicial y obtener puntos de vista sobre su funcionamiento en un entorno real, identificando ajustes necesarios antes de una expansión más amplia.

Una vez establecido y validado el sistema en el Departamento de Informática, se planifica su migración progresiva al resto de los departamentos de la Facultad de Ingeniería, con el objetivo de posicionarla como la primera facultad con una gestión digital de Instrumentos A dentro de la Universidad Nacional de Mar del Plata.

8.2.2 Plan de Estabilización y Escalabilidad Institucional

Se establece un "plan de estabilización" que contempla dos escenarios diferenciados según los resultados obtenidos durante la fase de implementación:

Escenario de expansión exitosa: Si el sistema demuestra estabilidad operativa y eficacia en toda la Facultad de Ingeniería, se contempla ofrecerlo como alternativa de gestión al resto de las unidades académicas de la Universidad Nacional de Mar del Plata. Esta expansión permitiría estandarizar la gestión de Instrumentos A a nivel institucional, generando beneficios de escala y homogeneización de procesos administrativos universitarios.

Escenario de mejora continua: En caso de identificarse limitaciones, ineficiencias o necesidades no contempladas durante las etapas de análisis y/o desarrollo, el feedback recibido será sistemáticamente analizado, documentado y priorizado para su incorporación en futuras actualizaciones del sistema. Este enfoque iterativo permitirá ajustar la plataforma a las necesidades reales de los usuarios y garantizará su evolución continúa respondiendo a los cambios en los procesos institucionales.

8.2.3 Modelo de Mantenimiento y Soporte Técnico

Se propone establecer un modelo de mantenimiento colaborativo que distribuya responsabilidades entre diferentes actores institucionales, asegurando la sostenibilidad técnica y funcional del sistema a largo plazo:

Responsabilidades del Centro de Cómputos de la UNMdP:

- Mantenimiento de la infraestructura tecnológica (servidores, base de datos, servicios auxiliares)
- Ejecución y monitoreo de la estrategia de backups automáticos
- Gestión de seguridad a nivel de infraestructura y actualización de componentes críticos del sistema
- Monitoreo del rendimiento, disponibilidad y estabilidad de la plataforma
- Resolución de incidencias técnicas relacionadas con la infraestructura

Responsabilidades del Departamento de Informática:

- Gestión funcional del sistema y coordinación con el Centro de Cómputos
- Soporte de primer nivel a usuarios finales de los diferentes departamentos
- Identificación, documentación y priorización de necesidades de mejoras funcionales
- Gestión de usuarios, roles y permisos desde la interfaz administrativa
- Capacitación a nuevos usuarios durante las fases de expansión del sistema
- Validación de actualizaciones y nuevas funcionalidades antes de su despliegue

Este modelo colaborativo entre el Centro de Cómputos y el Departamento de Informática asegura continuidad operativa, respuesta ágil ante incidencias técnicas, capacidad de evolución del sistema acorde a las necesidades institucionales cambiantes, y garantiza la sostenibilidad a largo plazo de la solución implementada.

8.2. Elementos No Contemplados Durante el Desarrollo

Durante el proceso de desarrollo e implementación del sistema, el equipo identificó diversos elementos y funcionalidades presentes en sistemas reales similares que no fueron incorporados en la solución final. Estas exclusiones obedecieron a tres factores principales: las limitaciones de alcance establecidas al inicio del proyecto, las restricciones temporales inherentes al cronograma académico, y aspectos que no fueron identificados y priorizados adecuadamente durante las fases iniciales de análisis y planificación.

8.2.1. Aspectos técnicos.

Redundancia y replicación de bases de datos

No se implementó un sistema de replicación automática de la base de datos que permitiera contar con réplicas sincronizadas en caso de fallo del servidor principal. Esta decisión se fundamentó en el análisis del contexto operativo específico del sistema: el volumen de datos manejado (aproximadamente 200 usuarios activos y entre 20-40 proyectos simultáneos anuales) y la criticidad moderada de la disponibilidad del sistema no justificaban la

complejidad técnica y administrativa que implica mantener una infraestructura de alta disponibilidad con réplicas de base de datos.

La estrategia adoptada se basó en backups completos automáticos mediante snapshots del servidor y de la base de datos, considerando que esta aproximación proporciona una protección adecuada para el contexto universitario. En el escenario de fallo catastrófico, la recuperación del sistema a partir de backups no representa un desafío técnico mayor y puede ejecutarse dentro de un tiempo de recuperación objetivo.

Arquitectura de alta disponibilidad y failover

No se diseñó ni implementó una arquitectura de servidores redundantes con capacidad de failover automático que garantizara continuidad del servicio ante fallas de hardware o software. Al igual que con la replicación de bases de datos, esta decisión respondió a un análisis de costo-beneficio considerando el alcance inicial del sistema.

Una infraestructura de alta disponibilidad no solo consume recursos computacionales significativamente mayores, sino que introduce complejidades administrativas considerables: gestión de balanceadores de carga, sincronización de estados entre servidores, monitoreo activo de salud de nodos, y procedimientos de recuperación automatizados. Para un sistema que opera principalmente en horario administrativo (lunes a viernes, 08:00-18:00 hs, durante 4 meses por año) y cuya interrupción temporal no genera impactos críticos en la operación institucional, estos costos adicionales no se justificaban en la fase inicial del proyecto.

La estrategia adoptada prioriza la simplicidad operativa y el mantenimiento por parte del personal técnico de la Universidad, quienes pueden gestionar efectivamente una arquitectura monolítica tradicional sin requerir expertise especializado en sistemas distribuidos.

Sistema de logging y auditoría avanzada

El sistema de logging representa una deuda técnica significativa identificada durante el desarrollo. Aunque Laravel proporciona capacidades nativas de registro de eventos mediante su sistema de logging integrado, no se diseñó ni implementó desde las fases iniciales del proyecto una estrategia comprehensiva de logging que contemplara:

- Niveles diferenciados de severidad para eventos del sistema
- Registro estructurado de acciones críticas de usuarios
- Herramientas de consulta y análisis de logs
- Alertas automáticas ante eventos anómalos
- Retención y rotación de archivos de log

Esta omisión en la fase de análisis representa un aprendizaje fundamental para el equipo de desarrollo. La ausencia de un sistema de logging robusto limita la capacidad de diagnóstico

ante problemas operativos, dificulta la detección de intentos de acceso no autorizado, y reduce la trazabilidad de acciones críticas para fines de auditoría.

Si bien el sistema actualmente implementa logging básico a nivel de aplicación y se aprovechan las capacidades de trazabilidad a nivel de base de datos (timestamps automáticos, eliminación lógica), la falta de una arquitectura de logging profesional constituye una mejora prioritaria para futuras versiones del sistema. Esta experiencia subraya la importancia de considerar aspectos de observabilidad, monitoreo y auditoría desde las fases tempranas de análisis y diseño, en lugar de incorporarlos reactivamente durante o después del desarrollo.

8.3. Consideraciones retrospectivas y oportunidades de mejora

Este apartado documenta las reflexiones surgidas durante y después del desarrollo del proyecto, identificando tecnologías alternativas, enfoques arquitectónicos y soluciones que podrían haber optimizado aspectos del sistema implementado.

El análisis retrospectivo contempla decisiones técnicas que, con la experiencia adquirida en el transcurso del proyecto, representan oportunidades de mejora para futuras iteraciones o desarrollos similares.

8.3.2. Tecnologías

Decidimos incorporar este apartado para documentar tecnologías o soluciones que, durante o después del desarrollo, identificamos como oportunidades de mejora o cambio respecto a lo ya implementado.

La selección de tecnologías estuvo parcialmente condicionada por las políticas internas de la institución, las cuales establecen el uso obligatorio de Laravel, PHP y MariaDB como parte de su stack tecnológico aprobado.

En cuanto a las tecnologías de frontend y herramientas complementarias, no se registraron restricciones institucionales al respecto, lo que permitió adoptar las soluciones más adecuadas a los requerimientos del problema abordado. En este marco, se optó por Next.js para el desarrollo del frontend y por WebSocket sobre Node.js para la gestión de comunicación en tiempo real.

Arquitectura de frameworks fullstack

Este apartado resulta crucial dado que aborda las tecnologías base utilizadas para el desarrollo del software: **Laravel** y **Next.js**. Ambos son frameworks de desarrollo fullstack con sus respectivos pros y contras.

El problema fundamental surge de la división de responsabilidades: la lógica frontend (incluyendo el renderizado por servidor) ocurre en Next.js, mientras que la lógica de servidores y API se gestiona en Laravel. Esta separación representa, en cierto punto, un **desperdicio de recursos**, ya que el stack tecnológico de ambos frameworks es considerablemente más amplio que lo requerido para las secciones específicas elegidas.

Limitaciones identificadas

Un caso particular de esta problemática se evidencia en la **implementación de WebSockets**. Laravel no posee soporte nativo para WebSockets, lo que nos obligó a crear un servidor WebSocket independiente utilizando tecnologías adicionales. En contraste, Next.js cuenta con soporte nativo para esta funcionalidad, lo que habría simplificado significativamente la implementación.

Impacto en el desarrollo

Si se hubiera realizado el desarrollo dentro del **mismo ecosistema tecnológico**, los siguientes aspectos se habrían visto beneficiados:

- **Tiempos de integración**: reducción de la complejidad en la comunicación entre frontend y backend
- **Tiempos de desarrollo**: eliminación de la necesidad de configurar y mantener múltiples entornos
- **Comunicación entre capas**: simplificación de la transferencia de datos y estados
- **Mantenimiento**: unificación del stack tecnológico y sus dependencias

Esta reflexión retrospectiva sugiere que una arquitectura monolítica o basada en un único framework fullstack habría optimizado el proceso de desarrollo y reducido la sobrecarga tecnológica del proyecto.

8.3.3. Alcance del backend y frontend

Como se mencionó en la sección 6.2.2, el desarrollo del backend se dividió en dos etapas: desarrollo y desatomización. Esta división fue consecuencia directa de la **diferencia de tiempos** en el desarrollo del backend y el frontend.

Si ambas secciones hubieran iniciado **simultáneamente**, o se hubiera realizado un mayor énfasis en el **diseño previo de la comunicación** entre capas, se podría haber evitado una etapa completa del desarrollo backend, o al menos reducido significativamente el tiempo invertido.

Oportunidades de mejora identificadas

- **Planificación sincronizada**: inicio paralelo del desarrollo frontend y backend
- **Diseño de contratos de API**: definición temprana y detallada de los endpoints y estructuras de datos
- **Comunicación entre equipos**: mayor coordinación en las especificaciones técnicas desde las etapas iniciales
- **Arquitectura previa**: establecimiento de un diseño de comunicación robusto antes de la implementación

Esta desincronización generó trabajo adicional en la fase de desatomización, donde fue necesario ajustar y refactorizar componentes que podrían haberse diseñado correctamente desde el principio con una mejor coordinación inicial.

8.3.4. Uso de bases de datos NoSQL

Existen soluciones implementadas dentro de SQL que podrían haberse beneficiado de un enfoque NoSQL, tales como los **campos secundarios de las materias** y el **registro de roles de usuarios**. Estos son campos dinámicos cuya estructura se adecúa mejor en formato JSON y no requieren la rigidez estructural ni las restricciones propias de las bases de datos relacionales.

Casos de uso identificados

Las siguientes estructuras de datos presentan características que las hacen candidatas ideales para almacenamiento NoSQL:

- **Campos secundarios de materias**: información variable y no estandarizada que cambia según el tipo de materia
- **Roles de usuarios**: permisos y capacidades que pueden variar dinámicamente según el contexto
- **Metadatos flexibles**: atributos opcionales que no todos los registros requieren

Ventajas de un enfoque híbrido

La implementación de una **base de datos híbrida** (SQL + NoSQL) habría proporcionado:

- **Flexibilidad estructural**: adaptación natural a esquemas de datos variables sin necesidad de migraciones frecuentes
- **Mejor rendimiento**: consultas optimizadas para datos no relacionales y documentos JSON
- **Simplificación del código**: eliminación de la necesidad de serializar/deserializar JSON en campos de texto

- **Escalabilidad**: mejor manejo de estructuras de datos que crecen horizontalmente

Tecnologías alternativas consideradas

Soluciones como **MongoDB**, **Redis** (para datos clave-valor) o incluso **PostgreSQL con JSONB** habrían ofrecido un balance adecuado entre la estructura relacional necesaria para datos críticos y la flexibilidad requerida para campos dinámicos, manteniendo la integridad referencial donde fuera necesaria mientras se optimiza el almacenamiento de datos semi-estructurados.

8.3.5. Dinámica del equipo

Tal como se mencionó en la división de grupos, dos estudiantes se adecuaron al backend y uno al frontend. Esta idea en principio fue acertada: simplificó la resolución de cada área y permitió una comunicación fluida entre los responsables de cada capa del sistema.

Desbalance de carga de trabajo

El problema surgió cuando el área de **frontend creció más de lo esperado**. El único estudiante que la desarrollaba comenzó a experimentar trabas en el desarrollo debido al volumen creciente de trabajo y la complejidad de las funcionalidades requeridas.

Estrategia de mitigación propuesta

Esta situación se hubiera podido mitigar implementando una **distribución más flexible de roles**, donde uno de los estudiantes dedicados al backend pivotara entre ambas áreas según la demanda de trabajo. Al tener conocimiento de las dos áreas tecnológicas, este enfoque habría:

- **Aligerado los tiempos de desarrollo**: distribución más equitativa de la carga de trabajo
- **Reducido cuellos de botella**: evitar que una sola persona se convirtiera en punto crítico del proyecto
- **Mejorado la integración**: mayor comprensión de ambas capas facilitando la comunicación técnica
- **Aumentado la resiliencia del equipo**: menor dependencia de individuos específicos para áreas críticas

8.4. Conclusiones y aprendizaje

El desarrollo de este proyecto ha representado una experiencia formativa integral que excede el ámbito puramente técnico, abarcando aspectos metodológicos, organizacionales y de trabajo colaborativo.

Gestión de proyectos de mediana complejidad

Hemos adquirido experiencia en la **planificación, desarrollo e implementación** de un sistema de mediana complejidad, enfrentando desafíos reales que requirieron la toma de decisiones técnicas fundamentadas y la resolución de problemas emergentes durante el ciclo de desarrollo.

Trabajo en equipo y colaboración

El proyecto nos permitió desarrollar habilidades de **trabajo colaborativo**, incluyendo:

- **División efectiva de tareas**: asignación de responsabilidades según las capacidades y áreas de especialización de cada miembro
- **Comunicación técnica**: coordinación constante entre las diferentes capas del sistema (frontend, backend, base de datos)
- **Resolución conjunta de problemas**: enfrentamiento de obstáculos técnicos mediante el intercambio de conocimientos y experiencias

Dominio de tecnologías modernas

La implementación del sistema nos brindó experiencia práctica con tecnologías ampliamente utilizadas en la industria:

- **Laravel**: desarrollo de APIs RESTful, gestión de autenticación, manejo de bases de datos relacionales y arquitectura MVC
- **Next.js**: renderizado del lado del servidor (SSR), enrutamiento dinámico, optimización de rendimiento y desarrollo de interfaces modernas
- **Integración de ecosistemas**: comunicación entre frameworks diferentes y gestión de servicios complementarios

Reflexión final

Este proyecto no solo consolidó nuestros conocimientos técnicos, sino que también nos proporcionó una visión realista de los desafíos inherentes al desarrollo de software profesional, preparándonos para enfrentar proyectos de mayor envergadura con una base sólida de experiencia práctica y aprendizajes documentados.

9. Memoria del proyecto

Resumen de proyecto

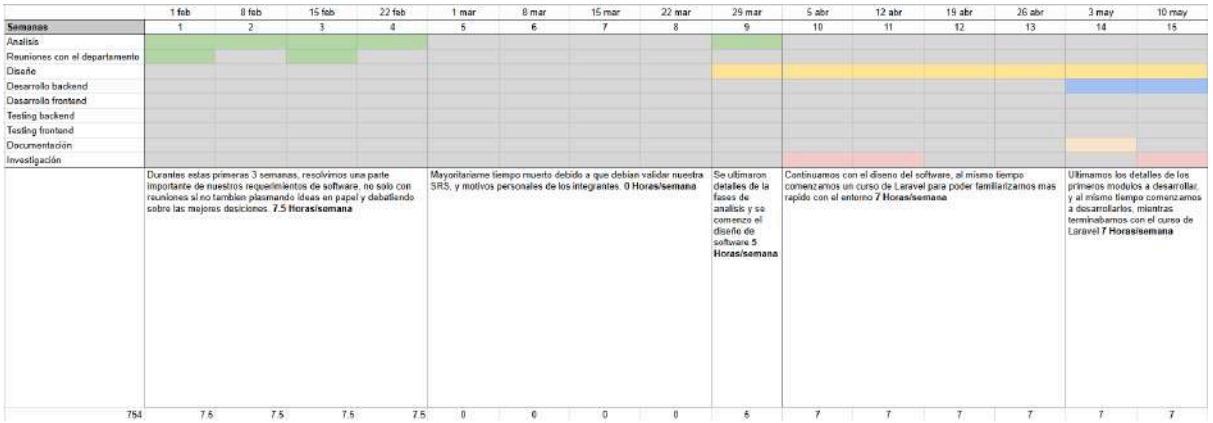
El presente proyecto, concebido con el objetivo de optimizar los procesos administrativos del cuerpo docente, fue ejecutado con el apoyo fundamental de la dirección del proyecto y el Departamento de Informática.

La ejecución del proyecto abarcó un período total de siete meses, distribuidos en actividades de análisis, diseño, investigación, desarrollo y pruebas (testing). Durante este ciclo de vida, la gestión del cronograma se reveló como un proceso dinámico y adaptativo.

En el transcurso de la ejecución, surgieron desafíos e inconvenientes no contemplados en la planificación inicial, atribuidos en gran medida a la curva de aprendizaje del equipo y a la subestimación de la complejidad técnica de ciertas tareas. Esta situación impuso la necesidad de reestructurar formalmente el cronograma y las fechas de finalización estimadas.

Incluso en etapas avanzadas donde la planificación parecía consolidada, la aparición de obstáculos técnicos emergentes obligó a realizar ajustes dinámicos sobre el planeamiento. A pesar de estas desviaciones respecto a la línea base, la capacidad de adaptación del equipo permitió el cumplimiento de los objetivos fundacionales, culminando en la entrega de un sistema maduro y completamente funcional.

Diagrama de Gantt en tiempo real de ejecución



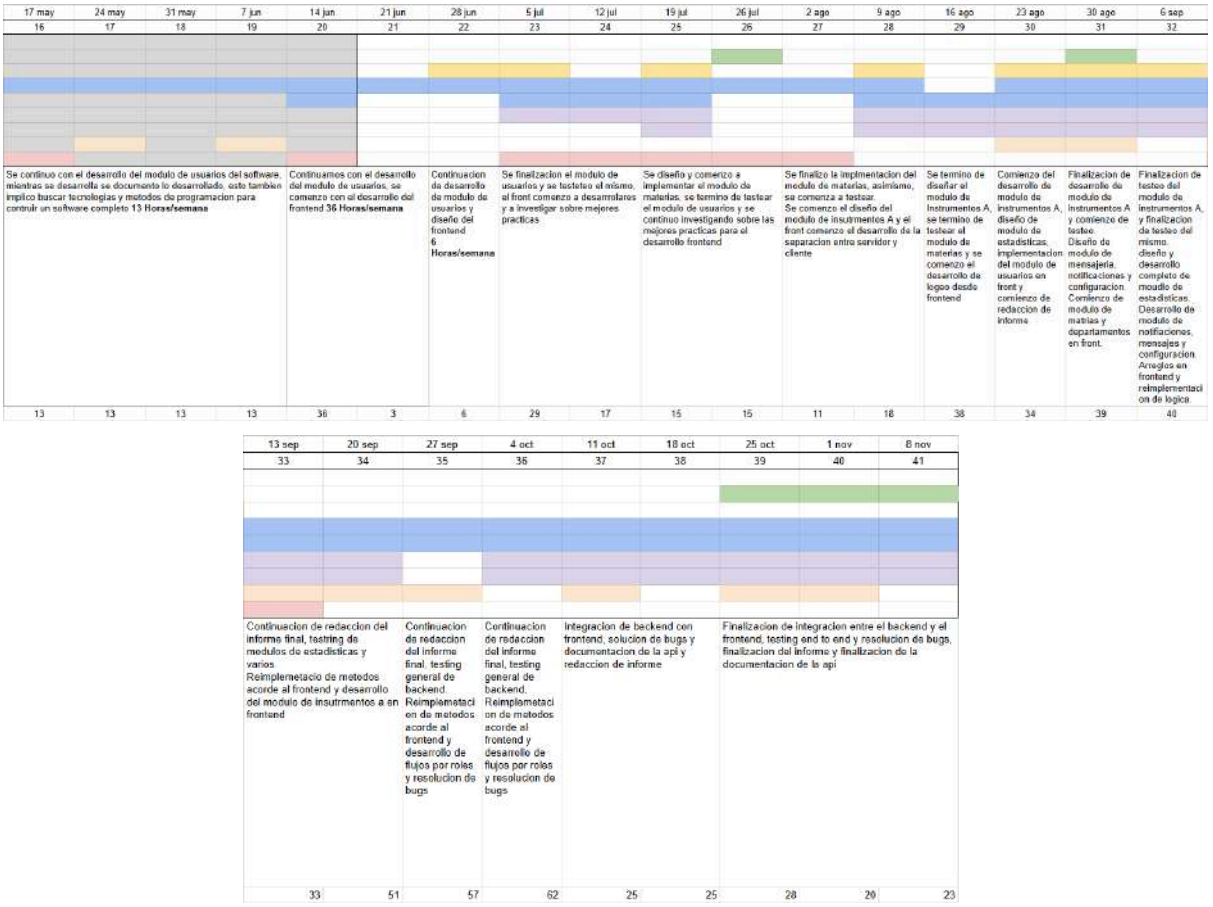


Figura 14. Diagrama de Gantt final

El cronograma de ejecución final del proyecto, detallado en el Diagrama de Gantt de la Figura 14 , evidencia una variación significativa respecto a la estimación refinada (presentada en la Figura 2). La planificación preveía una dedicación total de **650 horas-hombre**, mientras que la ejecución real documentada ascendió a **750 horas-hombre**.

Esto representa una **desviación neta de +100 horas**, lo que equivale a un **incremento del 15.38%** sobre el tiempo planificado.

El análisis retrospectivo de esta desviación permite identificar un conjunto de factores causales interrelacionados, los cuales no fueron dimensionados en su totalidad durante la fase de planificación:

- **Subestimación de la Complejidad y Curva de Aprendizaje:** La falta de experiencia previa del equipo en el desarrollo de proyectos de esta escala , sumada a la curva de aprendizaje asociada a las tecnologías empleadas, resultó en una subestimación del tiempo requerido para tareas clave. La **integración del sistema** se identificó como un punto crítico que consumió un volumen de horas superior al esperado.

- **Detección de Errores en Fases Tardías:** Una proporción de defectos (*bugs*) no fue identificada hasta las fases de integración (post-desarrollo). Esto generó un ciclo de **re-trabajo** no planificado, que implicó la reestructuración de código y la implementación de soluciones para garantizar la robustez frente a casos de uso críticos.
- **Costos de Comunicación y Definición de Interfaces:** Surgieron demoras atribuibles a la gestión de la comunicación técnica del equipo. Específicamente, la definición y ajuste de las **interfaces de comunicación** (contratos de la API) entre los componentes *frontend* y *backend* requirió más iteraciones de las previstas, generando fricción y retrasos en las tareas dependientes.
- **Gestión de Riesgos e Imprevistos:** La planificación original carecía de un *buffer* (colchón de tiempo) adecuado para absorber imprevistos, tanto técnicos (limitaciones tecnológicas descubiertas durante la implementación) como externos al proyecto (eventos personales).

.

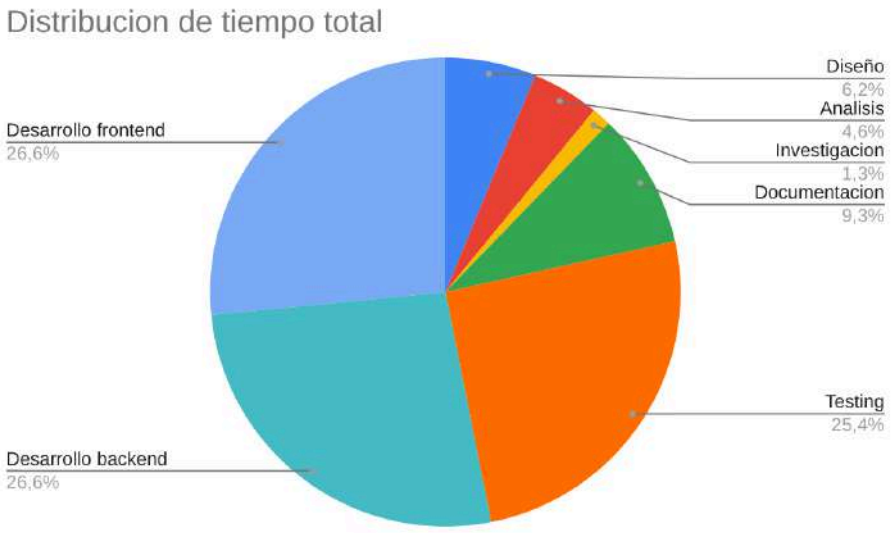


Figura 15. Diagrama de torta con distribución de tiempos

El análisis de la distribución porcentual del esfuerzo (Figura 15) permite extraer dos conclusiones fundamentales sobre la ejecución del proyecto:

1. **Validación de la Planificación Conceptual:** Se observa que el tiempo dedicado a las fases de **Análisis (4.6%)**, **Diseño (6.2%)** e **Investigación (1.3%)** fue minoritario frente a la ejecución. Esto demuestra que el estudio previo del problema fue exitoso y robusto, ya que la solución conceptual y la arquitectura de la solución no sufrieron variaciones significativas durante la fase de desarrollo, evitando refactorizaciones mayores.

2. **Impacto de la Curva de aprendizaje:** Como contraparte, la **"brecha tecnológica"** y los desafíos de **integración** consumieron una porción sustancial del tiempo de ejecución. Se estima que aproximadamente un **35% del 78.6%** dedicado al desarrollo y testing fue invertido en la resolución de estos conflictos, en lugar de en la implementación de nueva funcionalidad.

Si bien esta desviación representa un costo en horas-hombre, el equipo no la considera un resultado negativo. Dicha inversión de tiempo se tradujo en un aprendizaje práctico profundo sobre la resolución de errores complejos no anticipados y la aplicación de soluciones tecnológicas modernas bajo un escenario real.

10. Glosario

A

API (Application Programming Interface): Conjunto de reglas, definiciones y protocolos que permite que diferentes aplicaciones de software se comuniquen entre sí para intercambiar datos y funcionalidades de manera estandarizada y segura.

API REST (Representational State Transfer): Arquitectura de servicios web basada en el protocolo HTTP que establece un lenguaje de comunicación uniforme entre el cliente y el servidor mediante métodos estándar (GET, POST, PUT, DELETE).

Arquitectura Cliente-Servidor: Modelo de diseño donde las tareas se distribuyen entre proveedores de servicios (servidores) y solicitantes de servicios (clientes), separando responsabilidades y recursos.

Arquitectura Monolítica Modular: Enfoque que combina la simplicidad de una aplicación única desplegable con la organización estructural en módulos cohesivos y capas especializadas.

Autenticación: Proceso de verificar la identidad de un usuario o sistema mediante credenciales como contraseña, tokens o certificados digitales.

Autorización: Proceso que determina qué acciones puede realizar un usuario autenticado dentro del sistema según sus roles y permisos.

B

Backend: Capa del servidor que contiene la lógica de negocio, gestión de datos, servicios de red e infraestructura que opera de manera subyacente al sistema, no visible directamente para el usuario.

Backup (Copia de Seguridad): Proceso de crear y almacenar copias de los datos del sistema en una ubicación segura para poder restaurar la información ante pérdidas, errores o desastres.

Base de Datos SQL: Sistema de gestión que organiza información en tablas relacionadas mediante claves, permitiendo consultas complejas y garantizando integridad referencial.

C

Caché: Técnica de almacenamiento temporal de datos frecuentemente accedidos en una ubicación de acceso rápido, reduciendo tiempos de respuesta y carga del servidor.

Capa de Presentación: Componente de la arquitectura que maneja las interfaces de usuario y controladores, responsable de la interacción con el usuario final.

Capa de Acceso a Datos: Componente que abstrae y gestiona la persistencia de información, encapsulando las operaciones con la base de datos.

Capa de Lógica de Negocio: Componente que encapsula las reglas del dominio y procesos centrales del sistema, independiente de la presentación y persistencia.

Código de Estado HTTP: Número que indica el resultado de una solicitud HTTP (200 para éxito, 404 para no encontrado, 500 para error del servidor, etc.).

CORS (Cross-Origin Resource Sharing): Mecanismo de seguridad que controla qué dominios pueden acceder a recursos del servidor desde navegadores web.

CRUD: Acrónimo de las cuatro operaciones básicas en bases de datos: Create (Crear), Read (Leer), Update (Actualizar) y Delete (Eliminar).

CSR (Client-Side Rendering): Técnica donde el servidor envía HTML básico con JavaScript, y el navegador genera dinámicamente el contenido usando JavaScript en el cliente.

D

DTO (Data Transfer Object): Patrón de diseño que encapsula y estructura información transferida entre diferentes capas del sistema, controlando qué datos se exponen en cada contexto.

Docker: Plataforma que permite crear, desplegar y ejecutar aplicaciones en contenedores, garantizando portabilidad y consistencia entre entornos.

E

Encriptación: Proceso de transformar información legible en un formato codificado que solo puede ser leído por quien posee la clave de descifrado.

Endpoint: URL específica de una API que representa un punto de acceso para realizar una operación particular sobre recursos del sistema.

F

Factory Pattern (Patrón Fábrica): Patrón de diseño que centraliza la responsabilidad de creación de objetos, encapsulando la lógica de instanciación según parámetros específicos.

Failover: Capacidad de un sistema para cambiar automáticamente a un servidor o sistema de respaldo cuando el principal falla.

Feature Tests: Pruebas que validan funcionalidades completas del sistema simulando el comportamiento de usuarios finales mediante peticiones HTTP.

Framework: Estructura de software que proporciona funcionalidades base, bibliotecas y convenciones para facilitar y estandarizar el desarrollo de aplicaciones.

Frontend: Capa de presentación de una aplicación con la que el usuario interactúa directamente, incluyendo interfaces visuales y lógica de interacción.

Fullstack: Término que describe tecnologías o desarrolladores que trabajan tanto en frontend como backend, manejando todas las capas del sistema.

G

Gantt (Diagrama de): Herramienta de gestión de proyectos que representa tareas a lo largo del tiempo mediante barras horizontales, mostrando dependencias y duración.

Git: Sistema de control de versiones distribuido que permite rastrear cambios en código fuente y facilitar colaboración entre desarrolladores.

GitHub: Plataforma de alojamiento de código basada en Git que proporciona herramientas de colaboración, revisión y gestión de proyectos.

H

Hashing: Proceso de transformar datos mediante algoritmos matemáticos en cadenas de longitud fija que no pueden revertirse, usado para almacenar contraseñas de forma segura.

HTTP (HyperText Transfer Protocol): Protocolo de comunicación que define cómo se estructuran y transmiten mensajes entre cliente y servidor en la web.

I

Infraestructura: Conjunto de recursos tecnológicos (servidores, redes, almacenamiento) que soportan el funcionamiento de aplicaciones y servicios.

Instancia: Objeto específico creado a partir de una clase, con valores concretos para sus atributos y comportamientos definidos.

Interfaz: Contrato que define un conjunto de métodos que una clase debe implementar, sin especificar cómo se implementan.

Inyección de Dependencias: Patrón de diseño donde las dependencias de un objeto se proporcionan externamente en lugar de ser creadas internamente.

J

JSON (JavaScript Object Notation): Formato ligero de intercambio de datos basado en texto, fácil de leer para humanos y parsear para máquinas.

JWT (JSON Web Token): Estándar para crear tokens de acceso que permiten validar identidad de usuarios en sistemas stateless, encapsulando información relevante.

K

Kanban: Metodología ágil que visualiza el flujo de trabajo mediante tableros con columnas representando etapas del proceso, facilitando gestión y seguimiento.

Kubernetes: Sistema de orquestación de contenedores que automatiza despliegue, escalado y gestión de aplicaciones contenedorización.

L

Laravel: Framework PHP de código abierto que sigue el patrón MVC, proporcionando herramientas robustas para desarrollo de aplicaciones web modernas.

Logging: Proceso de registrar eventos, errores y actividades del sistema en archivos de log para debugging, auditoría y monitoreo.

M

MariaDB: Sistema de gestión de bases de datos relacionales de código abierto, fork de MySQL con mejoras de rendimiento y características adicionales.

Middleware: Componente que actúa como intermediario en el flujo de procesamiento de peticiones, implementando lógica transversal como autenticación o validación.

Migración (Base de Datos): Archivo versionado que define cambios estructurales en la base de datos, permitiendo evolución controlada del esquema.

Mock: Objeto simulado que imita el comportamiento de componentes reales, usado en pruebas para aislar código sin depender de servicios externos.

Modelo-Vista-Controlador (MVC): Patrón arquitectónico que separa la aplicación en tres componentes: Modelo (datos), Vista (presentación) y Controlador (lógica de control).

Modularidad: Principio de diseño que divide un sistema en módulos independientes con responsabilidades específicas, facilitando mantenimiento y reutilización.

N

Next.js: Framework React que proporciona renderizado del lado del servidor (SSR), generación estática y optimizaciones automáticas de rendimiento.

Node.js: Entorno de ejecución de JavaScript del lado del servidor, permitiendo construir aplicaciones de red escalables y eficientes.

NoSQL: Categoría de bases de datos que no siguen el modelo relacional tradicional, incluyendo documentos, clave-valor, grafos y columnas.

P

Patrón de Diseño: Solución reutilizable y probada a problemas comunes de diseño de software, estableciendo mejores prácticas de desarrollo.

Patrón Repository: Patrón que encapsula la lógica de acceso a datos, actuando como intermediario entre la lógica de negocio y la persistencia.

Patrón Service Container: Patrón que gestiona la creación y resolución de dependencias mediante un contenedor centralizado de servicios.

Patrón Strategy: Patrón que define familia de algoritmos encapsulados e intercambiables, permitiendo que varíen independientemente de los clientes.

Pest PHP: Framework de testing para PHP que proporciona sintaxis expresiva y legible para escribir pruebas unitarias y de integración.

PHP: Lenguaje de programación de código abierto especialmente diseñado para desarrollo web del lado del servidor.

Policy: Componente de autorización que encapsula lógica de permisos y privilegios, validando si un usuario puede realizar acciones específicas.

Polling: Técnica donde el cliente consulta repetidamente al servidor para verificar actualizaciones, menos eficiente que WebSockets.

R

RBAC (Role-Based Access Control): Modelo de control de acceso donde permisos se asignan a roles y usuarios se asignan a roles.

Redis: Sistema de almacenamiento en memoria de alto rendimiento usado como base de datos, caché y broker de mensajes.

Refactoring: Proceso de reestructurar código existente sin cambiar su comportamiento externo, mejorando legibilidad y mantenibilidad.

RefreshDatabase: Trait de Laravel que resetea la base de datos entre tests, garantizando aislamiento y resultados consistentes.

Repository: Clase que encapsula la lógica de acceso a datos, proporcionando interfaz abstracta para operaciones de persistencia.

S

Schedule (Cron Jobs): Sistema de programación de tareas que ejecuta comandos o scripts automáticamente en intervalos definidos.

Scope (Alcance): Límite definido de funcionalidades, características y entregas de un proyecto de software.

Scrum: Marco de trabajo ágil que organiza el desarrollo en ciclos iterativos fijos (sprints), empleando roles y ceremonias específicas para optimizar la entrega de valor y la productividad.

Servidor: Computadora o software que proporciona servicios, recursos o datos a otros sistemas llamados clientes.

SHA-256: Algoritmo criptográfico de hash que genera cadenas de 256 bits, usado para encriptación segura de contraseñas.

SQL (Structured Query Language): Lenguaje estándar para gestionar y consultar bases de datos relacionales.

Stakeholder: Persona, grupo u organización con interés en el proyecto y sus resultados, incluyendo usuarios, patrocinadores y directivos.

Stateless: Arquitectura donde cada petición contiene toda la información necesaria, sin mantener estado de sesión en el servidor.

T

Testing: Proceso de evaluar software para verificar cumplimiento de requisitos e identificar defectos antes del despliegue.

Timestamp: Marca de tiempo que registra fecha y hora exacta de un evento, creación o modificación de datos.

Trazabilidad: Capacidad de seguir y documentar el historial completo, ubicación y aplicación de elementos a lo largo de su ciclo de vida.

TTL (Time To Live): Período de tiempo durante el cual datos en caché son considerados válidos antes de requerir actualización.

TypeScript: Superconjunto de JavaScript que añade tipado estático, mejorando detección de errores y mantenibilidad del código.

U

Unit Tests: Pruebas que verifican componentes individuales aislados del sistema, asegurando funcionamiento correcto de cada unidad.

URL (Uniform Resource Locator): Dirección única que identifica un recurso en Internet, comúnmente conocida como dirección web.

V

Versionado: Gestión sistemática de cambios en código fuente, documentación o configuraciones mediante sistemas de control de versiones.

W

WebSocket: Protocolo de comunicación que establece canal bidireccional persistente entre cliente y servidor para comunicación en tiempo real.

WIP (Work In Progress): Límite de trabajo simultáneo en metodologías ágiles para evitar sobrecarga y mejorar calidad de entregas.

11. Anexos

Anexo I

Con el objetivo de proporcionar una referencia técnica completa y facilitar la integración con el sistema, se desarrolló un sitio web dedicado a la documentación de la API. Esta solución fue diseñada acorde a las necesidades específicas del proyecto, contemplando los requisitos particulares del dominio y las características de la arquitectura implementada.

La documentación técnica se encuentra disponible públicamente en el siguiente enlace: https://gia-api-documentation.netlify.app/

Anexo II

Elicitación de requerimientos

La elicitación de requerimientos constituye una de las fases más críticas en el ciclo de vida del desarrollo de software, estableciendo los fundamentos técnicos y funcionales que determinarán tanto la viabilidad como el éxito del proyecto. Esta etapa define no solamente los aspectos operativos del sistema a desarrollar, sino también las restricciones tecnológicas, organizacionales y regulatorias que condicionarán todas las decisiones posteriores de diseño e implementación.

El proceso de relevamiento de requerimientos se estructuró mediante una metodología mixta que combinó técnicas formales e informales de recopilación de información, garantizando la captura integral de necesidades tanto explícitas como implícitas de los stakeholders involucrados.

Metodologías de elicitación implementada

Supervisión académica especializada: La elicitación se desarrolló bajo la supervisión directa de los directores del proyecto, Mg. Spinelli, Adolfo Tomás y Lic. Rico, Carlos Alberto, quienes aportaron su experiencia profesional y conocimiento del dominio académico para orientar el proceso de identificación y priorización de requerimientos.

Sesiones de relevamiento estructuradas: Se implementaron reuniones semanales de menos de 90 minutos con los directores del proyecto, complementadas con sesiones específicas de validación y refinamiento de requerimientos. Estas reuniones siguieron una agenda estructurada que incluía revisión de requerimientos previamente identificados, presentación de nuevas necesidades emergentes y validación de la factibilidad técnica de los requerimientos propuestos.

Proceso iterativo de validación: Se estableció un ciclo continuo de revisión y ajuste de requerimientos, permitiendo la incorporación de modificaciones y precisiones durante todo el período de análisis, asegurando que el alcance del proyecto permaneciera alineado con las expectativas institucionales y las capacidades técnicas del equipo de desarrollo.

Fuentes de requerimientos identificadas

El análisis permitió identificar múltiples fuentes de requerimientos que condicionan el desarrollo del sistema:

Requerimientos del dominio del problema: Necesidades específicas derivadas del proceso actual de gestión de Instrumentos A, incluyendo flujos de trabajo, roles organizacionales, estados documentales y criterios de aprobación institucional.

Restricciones institucionales: Limitaciones tecnológicas establecidas por la Universidad Nacional de Mar del Plata y la Facultad de Ingeniería, incluyendo políticas de seguridad informática, estándares de desarrollo, infraestructura disponible y protocolos de acceso a sistemas institucionales.

Requerimientos regulatorios: Cumplimiento de normativas académicas específicas relacionadas con la documentación de asignaturas, procesos de acreditación y auditoría institucional, así como regulaciones de protección de datos personales y confidencialidad académica.

Requerimientos de escalabilidad: Consideraciones técnicas para la futura expansión del sistema hacia otras unidades académicas de la universidad, incluyendo flexibilidad arquitectónica, modularidad del sistema y capacidad de adaptación a diferentes procesos organizacionales.

Técnicas de elicitación empleadas

Entrevistas semiestructuradas: Conversaciones dirigidas con stakeholders clave para identificar necesidades específicas, expectativas del sistema y restricciones operativas no documentadas.

Análisis de procesos existentes: Estudio detallado del flujo actual de gestión de Instrumentos A, identificando puntos de fricción, cuellos de botella y oportunidades de mejora a través de la digitalización.

Revisión documental: Análisis de formularios, procedimientos y normativas institucionales existentes para comprender los requerimientos formales del proceso de acreditación.

Documentación y trazabilidad

Todos los requerimientos identificados fueron documentados siguiendo estándares de ingeniería de software, incluyendo:

- **Identificación única:** Cada requerimiento recibió un código alfanumérico específico (RF01, RNF01, etc.) para facilitar su trazabilidad durante todo el ciclo de desarrollo
- **Criterios de aceptación:** Definición específica de condiciones que debe cumplir cada requerimiento para considerarse satisfactoriamente implementado
- **Prioridad y dependencias:** Clasificación de requerimientos según su criticidad y identificación de relaciones de dependencia entre funcionalidades

Esta metodología estructurada de elicitación garantizó la identificación integral de requerimientos, minimizando riesgos de omisiones críticas y estableciendo bases sólidas para las etapas posteriores de diseño e implementación del sistema

Requerimientos funcionales

RF01: Login de usuario

- El sistema debe proporcionar un mecanismo de autenticación que permita a los usuarios iniciar sesión mediante la validación de credenciales compuestas por nombre de usuario y contraseña.

RF02: Validación del usuario

- El sistema deberá validar las credenciales ingresadas por el usuario.
- En caso de fallar 3 veces de manera consecutiva, el usuario será forzado a cambiar su contraseña.

RF03: Olvido de contraseña

- El sistema deberá proporcionar una funcionalidad de recuperación de contraseña que envíe un enlace de restablecimiento al correo electrónico del usuario cuando éste lo solicite.

RF04: Logout de usuario

- El sistema debe permitir que los usuarios cierren su sesión activa de manera segura, finalizando su acceso a las funcionalidades del sistema.

RF05: Sistema de mensajería y notificaciones

- El sistema deberá implementar un mecanismo de comunicación bidireccional que mantenga un registro persistente de los mensajes intercambiados.

- Debe incluir los metadatos de identificación del remitente y destinatario, además de la fecha y hora de cada mensaje.

RF06: Modificación de datos personales no sensibles

- El sistema deberá exigir que en el primer acceso, los usuarios modifiquen de manera obligatoria su nombre de usuario y contraseña.
- Debe permitir la modificación posterior de datos personales, excluyendo información sensible, como el nombre, apellido y número de documento.

RF07: Consulta de lista de docentes

- El sistema deberá mantener y mostrar una lista actualizada de todos los docentes que se encuentren trabajando actualmente en la facultad.

RF08: Busqueda de docentes

- El sistema debe permitir al administrador y los directores realizar búsquedas de docentes utilizando cualquiera de sus atributos como criterio de filtrado.

RF09: Estadísticas del sistema

- El sistema debe generar reportes estadísticos que incluyan información sobre la cantidad de usuarios por rol, el número de docentes por departamentos, cantidad de instrumentos A por estado, entre otras.

RF10: Consulta del instrumento A

- El sistema debe permitir que los roles de administrador, secretaría académica, director y observador puedan consultar los instrumentos A creados por los docentes.
- Se deberá proporcionar la posibilidad de realizar comentarios en el documento.

RF11: ABMC de cargos en el departamento

- El sistema deberá permitir que los directores puedan crear, modificar y eliminar los cargos existentes dentro de su departamento.

RF12: Especificaciones de los cargos

- El director debe ingresar los datos completos de cada cargo, incluyendo las horas mínimas y máximas del mismo, y los créditos del cargo.

RF13: ABMC de docentes

- El sistema deberá permitirle al administrador dar de baja docentes, modificar sus datos personales y consultar la información almacenada.

RF14: Consulta del historial de instrumentos A

- El sistema deberá permitir a los directores y docentes acceder al registro completo donde se almacenan todos los instrumentos A creados hasta el momento de su respectivo departamento o materia..

RF15: Exportación de instrumento A hacia PDF

- El sistema deberá permitir la conversión a formato PDF de los instrumentos A bajo la nomenclatura "Año/Cuatrimestre/Materia".

RF16: Habilitación de los docentes a crear el instrumento A.

- Los directores deberán asignar permisos específicos a los docentes seleccionados a través de cargos para la creación del instrumento A.

RF17: Creación del instrumento A

- El sistema debe presentar al docente habilitado con distintas opciones para la creación del instrumento A, dando la posibilidad de rellenar los campos disponibles con la información de la materia o un instrumento A pasado.

RF18: Cargado de créditos del docente

- El sistema deberá calcular y asignar automáticamente los créditos a los docentes basándose en los cargos que posean dentro de la institución.
- El valor máximo admisible de créditos es 40.

RF19: Super Usuario

- El sistema deberá contar con un rol de administrador que posea privilegios amplios sobre la gestión de usuarios, roles, departamentos, materias y configuración del sistema.

RF20: Cambio de formato del instrumento A

- El sistema deberá permitir la capacidad de agregar y eliminar campos extras en el instrumento A a los docentes encargados de su creación.

RF21: Versionado del instrumento A

- En caso de que el instrumento A sea rechazado en alguna de las etapas de verificación, se le habilitará al docente titular la funcionalidad de crear una nueva versión del instrumento.

RF22: ABMC de cargos y asignaturas

- El sistema debe permitir realizar operaciones de alta, baja, modificación y consulta sobre la información de departamentos.

Requerimientos no funcionales

- **Rendimiento:** El sistema deberá soportar un flujo operativo diario que permita el procesamiento concurrente de al menos 100 usuarios, y garantizar la capacidad de gestionar simultáneamente operaciones de altas, modificaciones de datos e intercambio de mensajes sin degradación significativa en el rendimiento.

- **Interfaces externas:** El sistema poseerá una interfaz gráfica intuitiva y accesible que garantice la interacción efectiva de usuarios con distintos niveles de experiencia técnica para así lograr una rápida comprensión y operabilidad del sistema. Además, el sistema contará con un sistema de correo el cual estará encargado de informar a los usuarios de distintos eventos, como puede ser la asignación de un rol o la habilitación del instrumento A para su creación.

- **Almacenamiento:** El sistema almacenará toda la información de usuarios, metadatos, instrumentos A y mensajes en una base de datos relacional MariaDB, estos datos serán conservados durante toda la vida útil del sistema.

- **Restricciones de diseño:** El desarrollo del sistema se limitará al uso de las siguientes tecnologías: PHP, HTML, CSS, TypeScript y MariaDB, junto a los frameworks Laravel y Next.js, aparte del uso de librerías específicas. La infraestructura funcionará sobre un servidor y base de datos que proporciona la institución educativa.

- **Atributos de calidad del software:** El sistema debe estar disponible durante todo el ciclo académico universitario y utilizar técnicas de tolerancia a fallos para funcionar correctamente ante cualquier problema. Debe incluir mecanismos de autenticación sólidos para permitir el acceso solo a usuarios autorizados y proteger la información

personal y documentos mediante técnicas seguras de almacenamiento. El sistema de roles proporcionará confidencialidad entre usuarios, mientras que el código será legible para facilitar el mantenimiento futuro y reducir costos y tiempos.

Modelado del sistema

Entidades

El modelado de entidades constituye un componente fundamental en el diseño de sistemas de información, proporcionando una representación abstracta de los elementos del mundo real que interactúan dentro del dominio del problema. Este proceso de abstracción permite identificar, clasificar y estructurar los objetos de datos que el sistema debe gestionar, estableciendo las bases para el diseño de la base de datos y la arquitectura del software.

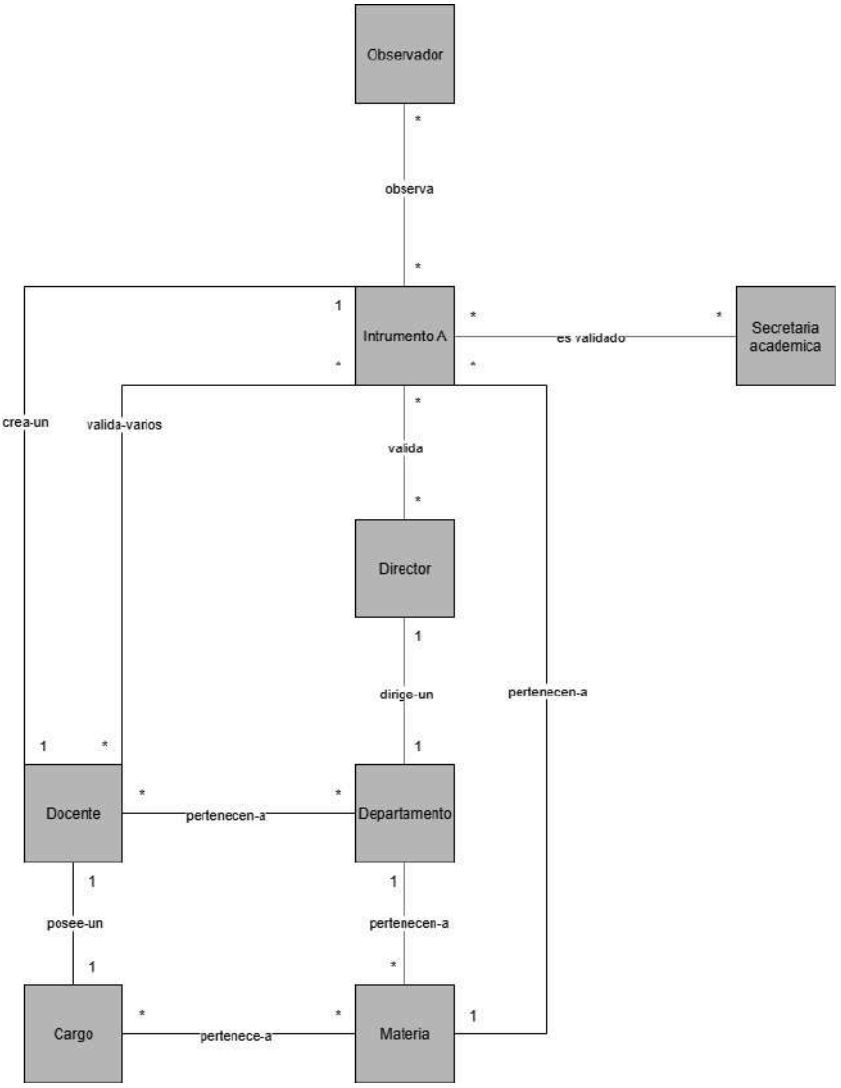


Figura 3. Diagrama de dominio

Identificación y clasificación de entidades

A través del análisis del dominio del problema (Figura 3) y los requerimientos funcionales identificados, se estableció un modelo de entidades que refleja fielmente la estructura organizacional y los procesos académicos involucrados en la gestión de Instrumentos A. El modelo contempla tanto entidades principales del negocio como entidades de soporte necesarias para el funcionamiento integral del sistema.

Entidades principales del sistema

Entidades de roles y usuarios:

- **Usuario:** Entidad base que representa a cualquier persona con acceso al sistema, conteniendo atributos comunes de identificación y autenticación.
- **Administrador:** Entidad especializada responsable de la gestión integral del sistema, configuración de parámetros y administración de usuarios.
- **Director:** Entidad que representa a los directores de departamento, con responsabilidades de supervisión y aprobación de Instrumentos A dentro de su jurisdicción.
- **Docente:** Entidad que modela a los profesores responsables de la creación y mantenimiento de Instrumentos A para las asignaturas a su cargo.
- **Observador:** Entidad que permite el acceso de consulta sin permisos de modificación, facilitando la supervisión y auditoría de procesos.
- **Secretaría Académica:** Entidad representativa del área administrativa responsable de la validación final y aprobación definitiva de Instrumentos A.

Entidades académicas y organizacionales:

- **Departamento:** Entidad que modela las unidades organizacionales de la facultad, estableciendo la estructura jerárquica y de responsabilidades académicas.
- **Materia:** Entidad que representa las asignaturas del plan de estudios, conteniendo información curricular específica y metadatos académicos.
- **Cargo:** Entidad que define las posiciones académicas y sus responsabilidades asociadas, estableciendo la relación entre docentes y materias.

Entidades documentales:

- **Instrumento A:** Entidad central del sistema que modela el documento académico objeto de gestión, incluyendo contenidos programáticos, estados de aprobación, versionado y trazabilidad del proceso de validación.

Relaciones entre entidades

El modelo de entidades establece un conjunto de relaciones que reflejan las interdependencias y flujos de información del proceso real:

Relaciones jerárquicas: Establecen la estructura organizacional mediante asociaciones entre Departamentos, Directores, Docentes y Cargos, respetando la cadena de autoridad institucional.

Relaciones de proceso: Modelan el flujo de trabajo del Instrumento A a través de los diferentes roles involucrados en su ciclo de vida, desde la creación hasta la aprobación final.

Relaciones de auditoría: Proporcionan trazabilidad completa de acciones, modificaciones y estados del Instrumento A, permitiendo la reconstrucción histórica del proceso de gestión.

Diagrama de entidad relación

Una vez identificadas las entidades del sistema y sus relaciones conceptuales, el siguiente paso es diseñar el diagrama entidad-relación (ER). Este diagrama representa la estructura lógica de los datos y define cómo las entidades se relacionan e interactúan entre sí dentro del sistema. El diagrama ER funciona como el plano arquitectónico de la base de datos, siendo la base para su posterior implementación en el esquema físico.

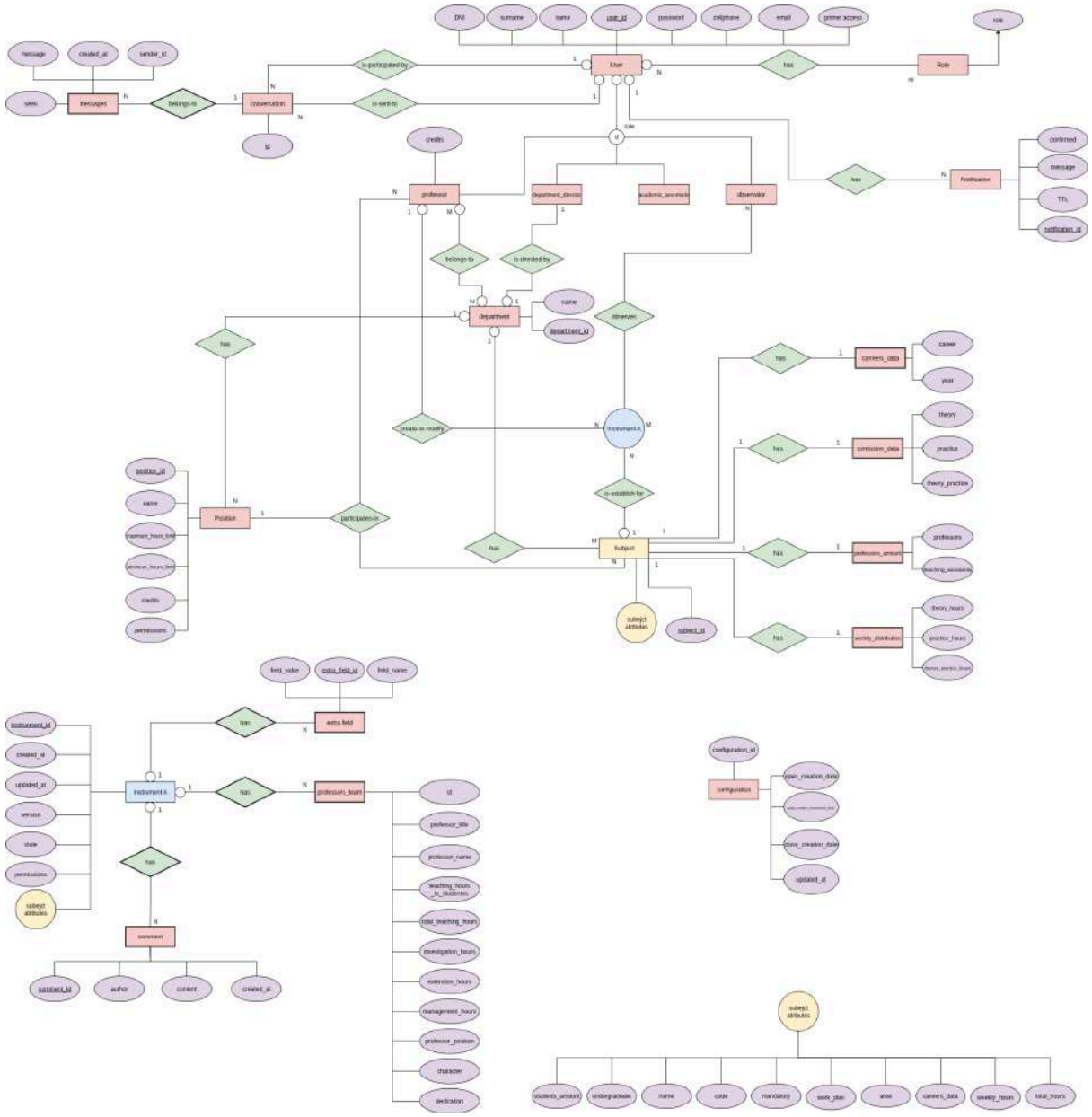


Figura 4. Diagrama de Entidad Relación completo

En la Figura 4 se muestra el diagrama entidad-relación integral del sistema, que ilustra la totalidad de entidades y sus respectivas interrelaciones. El diagrama permite visualizar la estructura modular del sistema y las conexiones entre sus componentes.

Módulo de usuarios y roles

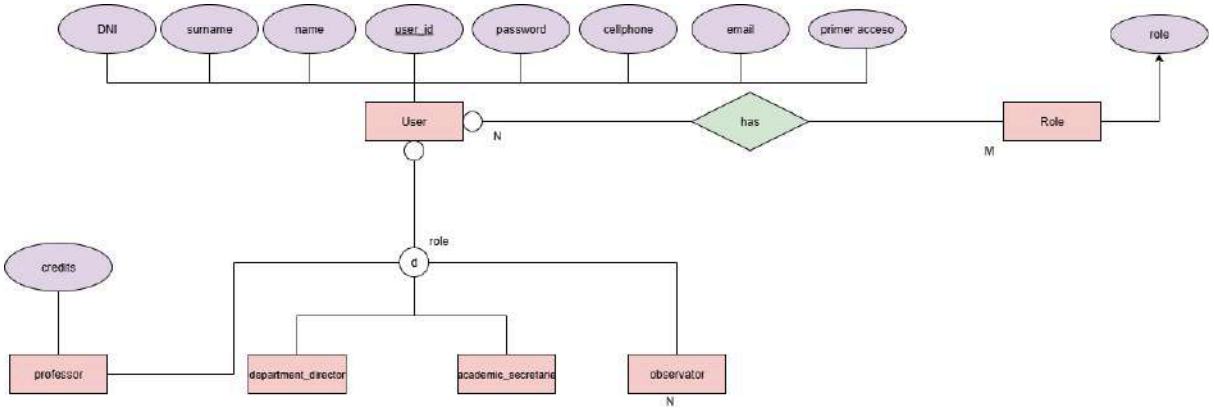


Figura 4.1. Diagrama de Entidad Relacion de Usuarios y Roles

En la Figura 4.1 se presentan las entidades Usuario y Rol junto con sus relaciones correspondientes. Se establece que un usuario puede tener ninguno, uno o múltiples roles asignados. Asimismo, se incluye una entidad intermedia que gestiona el registro de los roles asociados a cada usuario.

Módulo de mensajería y notificaciones

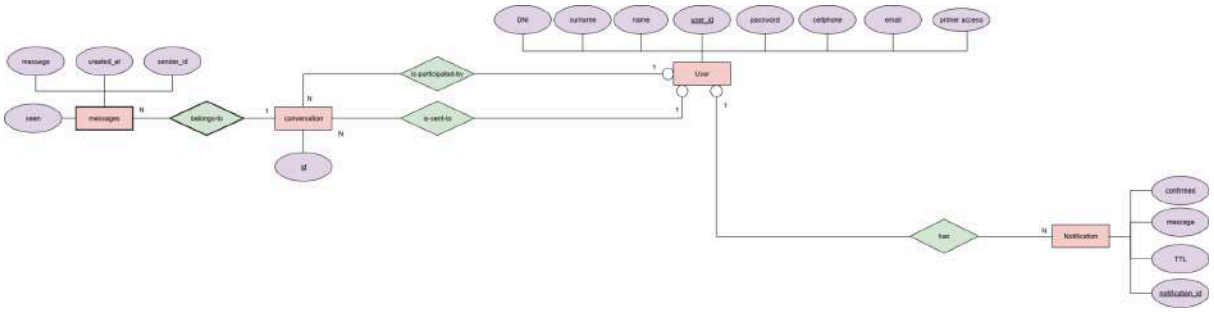


Figura 4.2. Diagrama de Entidad Relación de las Notificaciones y Conversaciones

En la Figura 4.2 se presenta la relación entre las entidades Notificación y Usuario, estableciendo una cardinalidad que permite a un usuario recibir ninguna, una o múltiples notificaciones. Asimismo, los usuarios participan en las conversaciones desempeñando los roles de emisor o receptor. Cada conversación está compuesta por los usuarios participantes y los mensajes intercambiados entre ellos.

Módulo de materias, cargos y departamentos

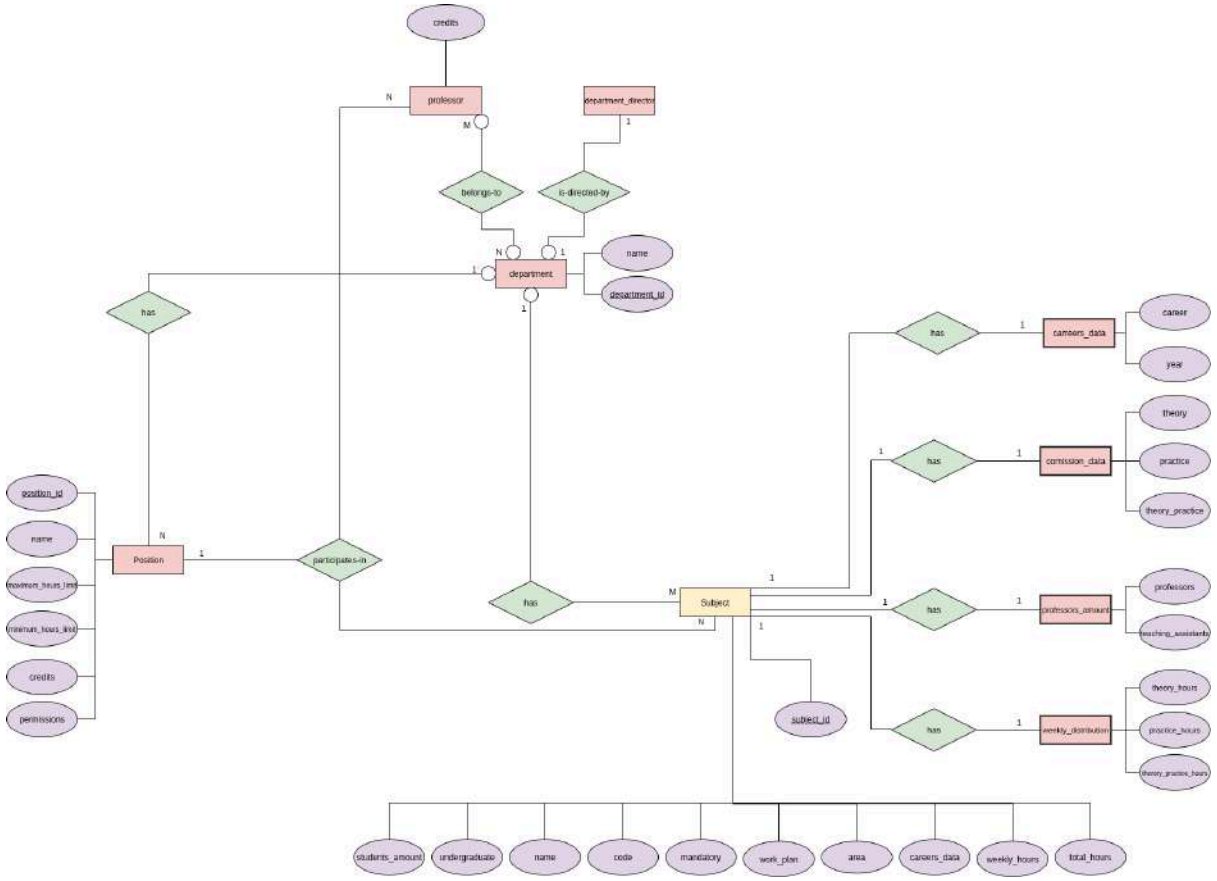


Figura 4.3. Diagrama de Entidad Relación de Materias, Cargos y Departamentos

En la Figura 4.3 se presentan las entidades correspondientes al módulo académico: Director, Departamento, Docente, Cargo y Materia. Se establece que cada director está asignado a un departamento específico, mientras que los docentes pueden pertenecer a ninguno, uno o más departamentos determinados, asimismo también pueden desempeñar diversos cargos en las materias impartidas por dicho departamento, las cuales se conforman de una serie de atributos y relaciones únicas como los datos de la comisión a la que pertenece, la cantidad de docentes y la distribución horaria semanal.

Módulo de instrumento A

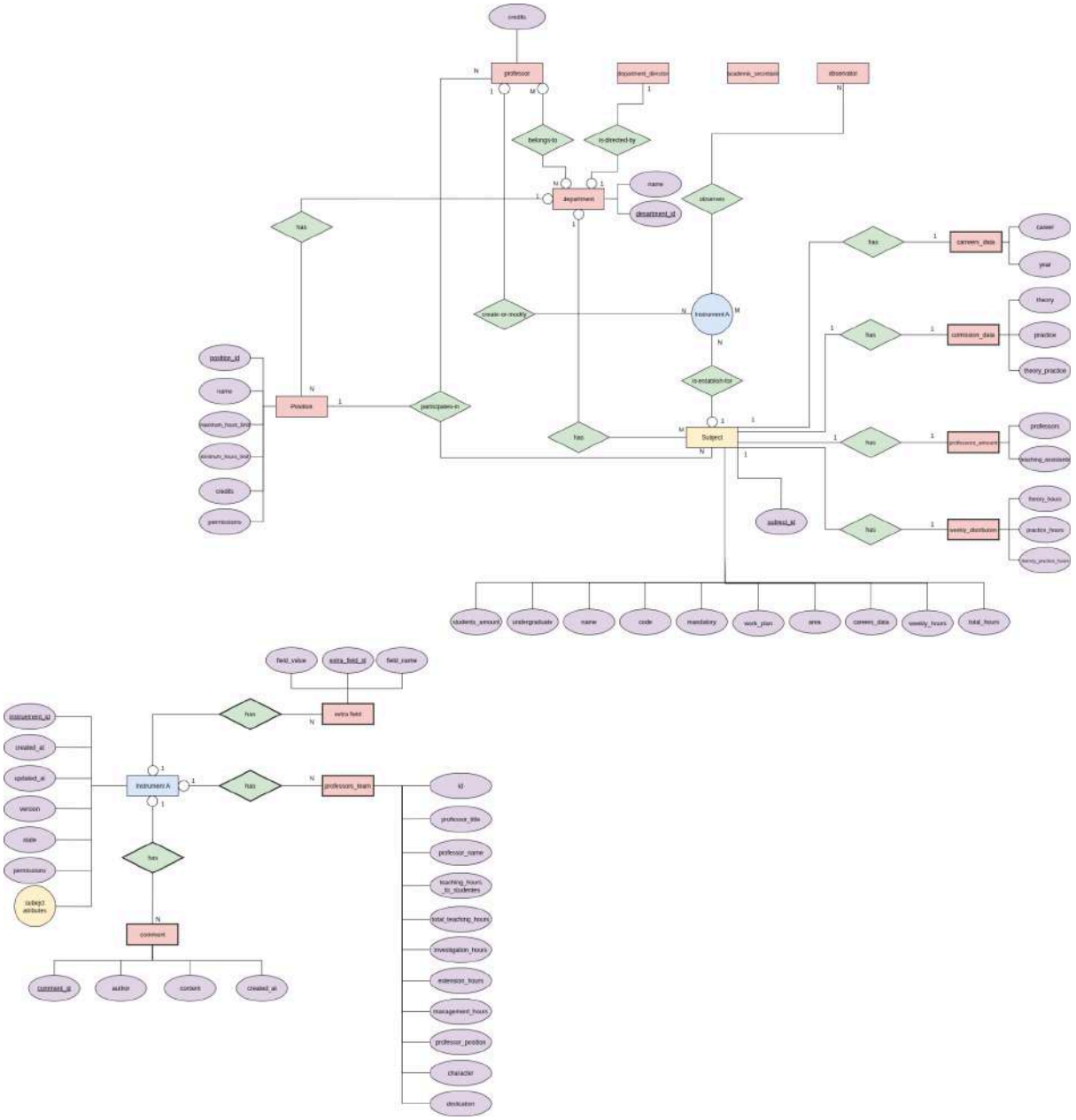


Figura 4.4. Diagrama de Entidad Relación de Instrumento A

En la Figura 4.4 se presenta la extensión del módulo académico mediante la incorporación de la entidad Instrumento A, la cual actúa como elemento integrador del sistema. Se establece que una materia puede generar múltiples instancias de Instrumento A a lo largo del tiempo, manteniendo una relación temporal evolutiva. El Instrumento A hereda los atributos de la entidad Materia e incorpora elementos específicos como equipo docente, campos adicionales y comentarios. Adicionalmente, este módulo introduce dos nuevos tipos de roles: Observador y Secretaria Académica, ampliando las funcionalidades del sistema de gestión.

Módulo de configuración

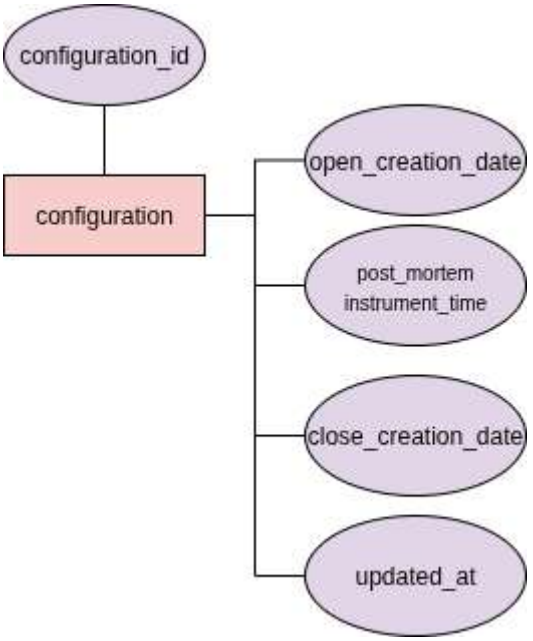


Figura 4.5. Diagrama de Entidad Relación de Configuración

En la Figura 4.5 se presenta el módulo de configuración del sistema, cuya función principal consiste en establecer las fechas de apertura y cierre para la creación de instrumentos A.

Modelo de casos de uso

El modelo de casos de uso constituye un artefacto fundamental en el proceso de desarrollo de software, ya que establece dos conceptos esenciales para la comprensión del sistema: los **actores** y los **casos de uso**.

Actores

Los actores representan a los usuarios finales del sistema desde una perspectiva de software, caracterizándose por encarnar roles específicos y reales dentro de la organización o institución. Estos roles definen las responsabilidades y permisos que cada tipo de usuario posee en relación con el sistema. De los cuales podemos encontrar:

- **Observadores**: Actores externos al proceso de creación de instrumentos de evaluación A, cuya función principal consiste en proporcionar retroalimentación objetiva durante el proceso de desarrollo. Su participación está sujeta a la discreción del docente responsable, permitiendo comentarios y sugerencias que enriquezcan la calidad del instrumento desde una perspectiva independiente.
- **Docentes**: Constituyen el núcleo operativo del sistema, siendo responsables de la creación, modificación y revisión inicial de los instrumentos A. Estos actores están

vinculados a materias específicas dentro de la estructura académica y poseen cargos definidos en las mismas. Su rol es fundamental ya que inician el flujo de validación del sistema.

- **Director de Departamento**: Actor que desempeña una función de enlace en la cadena administrativa, ejerciendo supervisión sobre el departamento, las materias, los cargos docentes y el personal académico asociado. Representa el primer nivel de validación formal en el proceso de aprobación de instrumentos A, posterior a su creación por parte de los docentes.
- **Secretaría Académica**: Constituye la instancia final de validación en el proceso de aprobación de instrumentos A. Este actor administrativo posee la autoridad para aprobar definitivamente la creación del instrumento o devolver el proceso a etapas anteriores para su revisión. Su intervención se limita exclusivamente a instrumentos que han alcanzado la fase final de validación.
- **Administrador del Sistema:** Actor técnico responsable de la configuración y mantenimiento del sistema, facilitando que los demás actores puedan ejecutar sus funciones de manera efectiva. Sus responsabilidades incluyen la creación de departamentos, configuración de materias y asignación de roles y permisos a los usuarios del sistema.

Usos del sistema

Los casos de uso describen las funcionalidades y comportamientos que el software debe proporcionar para satisfacer las necesidades de cada actor. En otras palabras, especifica qué acciones puede realizar cada tipo de usuario dentro del sistema y qué resultados se esperan de estas interacciones.
El sistema organiza sus funcionalidades en tres categorías principales de casos de uso, cada una respondiendo a diferentes niveles de interacción y permisos dentro de la plataforma.

Casos de Uso Administrativos

Comprenden las operaciones fundamentales para la configuración y mantenimiento de la estructura organizacional del sistema. Estas funcionalidades incluyen:

- **Gestión de Materias**: Creación, modificación, eliminación y consulta de materias académicas.
- **Administración de Departamentos**: Operaciones CRUD (Create, Read, Update, Delete) sobre la estructura departamental.
- **Gestión de Cargos**: Configuración y mantenimiento de los diferentes cargos académicos.
- **Asignación de Roles**: Vinculación de usuarios con roles específicos y sus permisos correspondientes dentro del sistema.

Casos de Uso de Gestión y Análisis de Instrumentos A

Engloban todas las operaciones directamente relacionadas con el ciclo de vida de los instrumentos de evaluación tipo A:

- **Gestión del Instrumento**: Creación, modificación y eliminación de instrumentos tipo A.
- **Sistema de Comentarios**: Funcionalidad para agregar observaciones y retroalimentación sobre los instrumentos.
- **Proceso de Validación**: Flujo de aprobación que sigue la jerarquía institucional establecida.
- **Consulta de Historial**: Acceso al registro histórico de instrumentos tipo A por materia, permitiendo trazabilidad y análisis temporal.

Casos de Uso Generales del Sistema

Constituyen las funcionalidades básicas que no requieren roles específicos para su ejecución y que facilitan la interacción general con la plataforma:

- **Autenticación**: Procesos de inicio y cierre de sesión en el sistema.
- **Comunicación**: Funcionalidades de envío y recepción de mensajes entre usuarios del sistema.

Esta clasificación permite una comprensión estructurada de las capacidades del sistema y facilita tanto el diseño como la implementación de las diferentes funcionalidades según el nivel de acceso requerido.

Diagrama de casos de uso

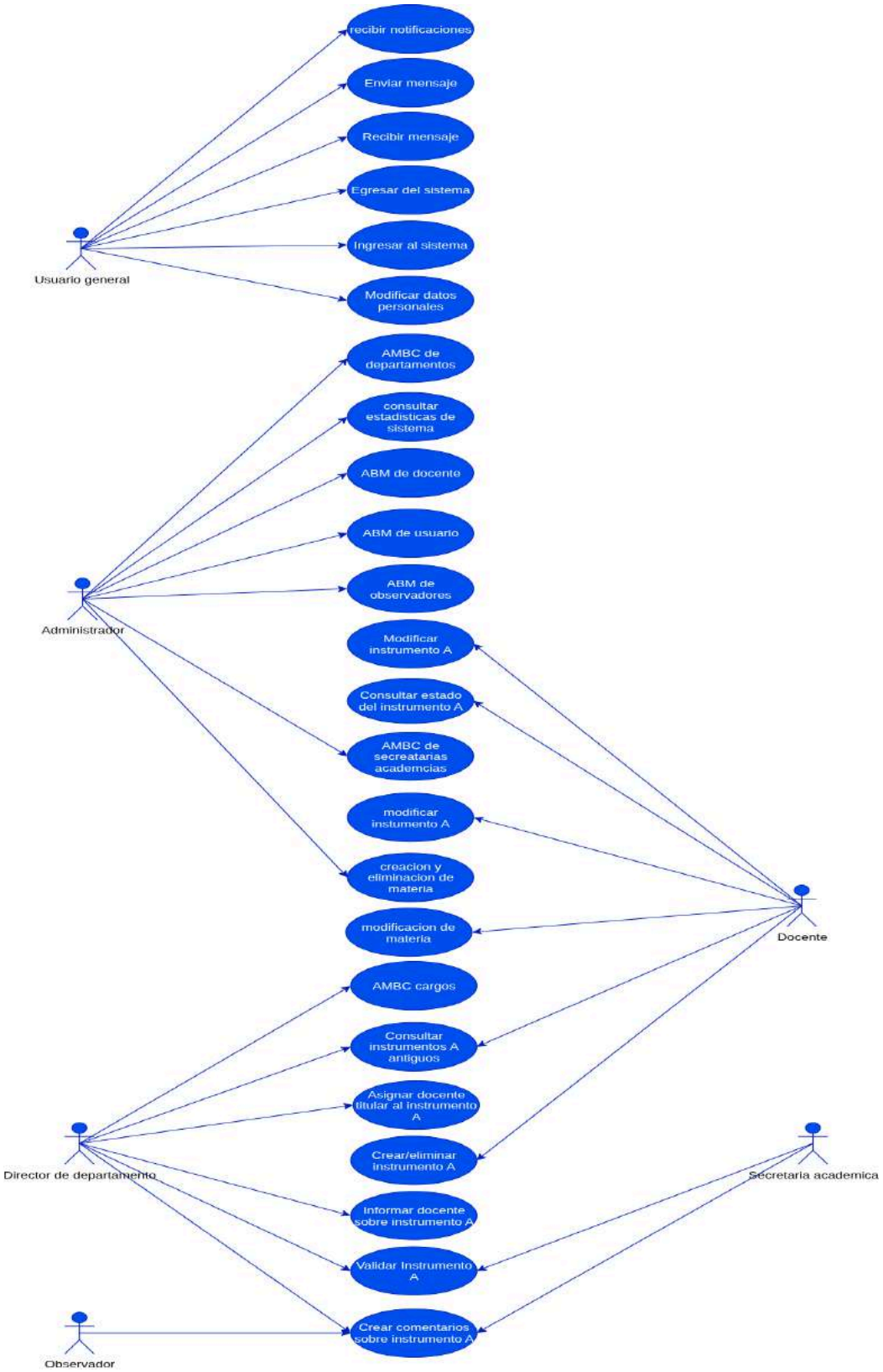


Figura 5. Diagrama de casos de uso

La relevancia de la figura 5 y el Modelo de casos de uso radica en su capacidad para establecer una comunicación clara entre los stakeholders del proyecto, definiendo de manera precisa tanto quién utilizará el sistema como qué funcionalidades específicas necesita cada tipo de usuario para cumplir con sus objetivos organizacionales.

Modelado de procesos del sistema

Esta sección tiene como objetivo explicar, mediante acciones sencillas y secuenciales, el funcionamiento operacional del sistema, detallando los flujos de trabajo que ejecutan los diferentes actores para cumplir con los objetivos del sistema.
La documentación de procesos complementa el análisis estructural presentado en las secciones anteriores, proporcionando una perspectiva dinámica del comportamiento del sistema en funcionamiento.

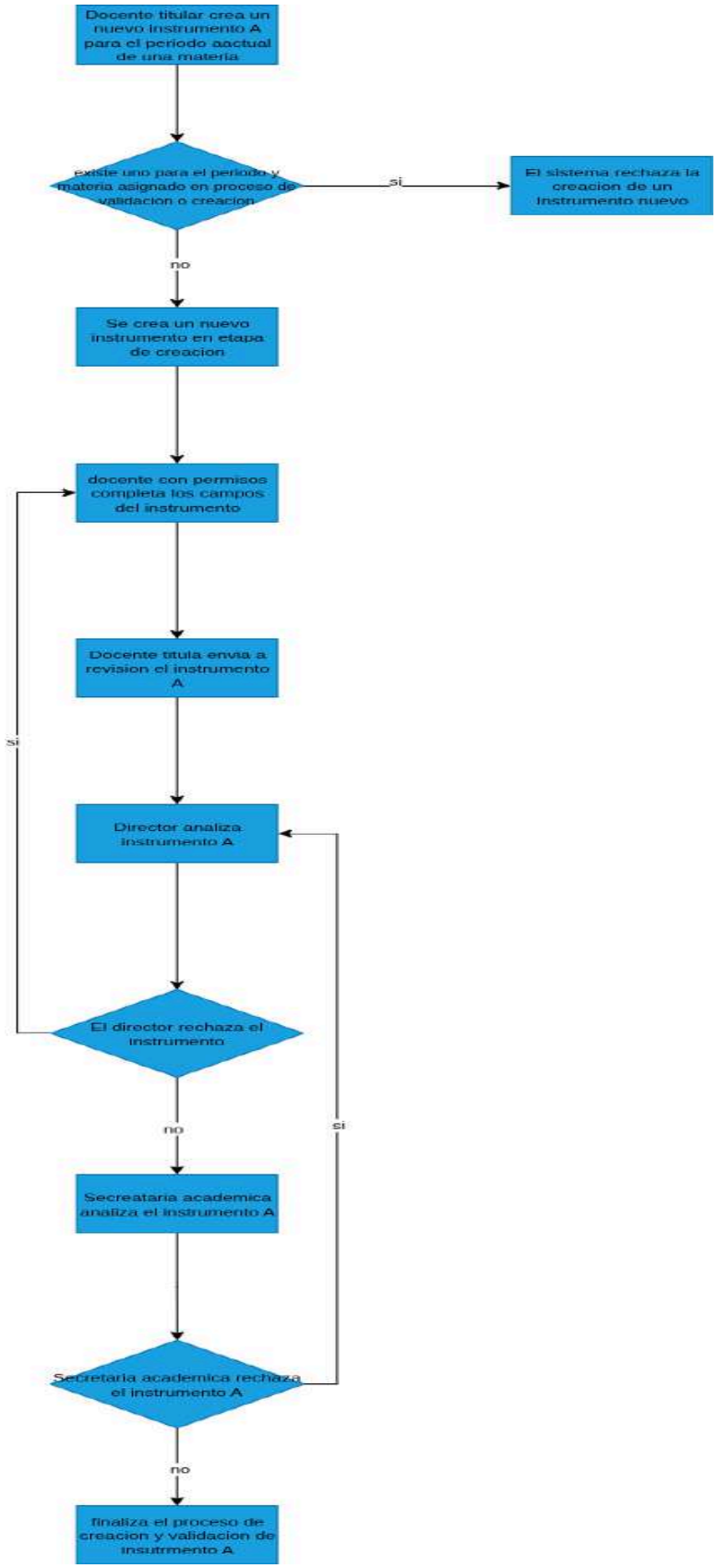


Figura 6. Diagrama de Flujo principal

El diagrama principal del sistema representa el flujo de validación de instrumentos tipo A, ilustrado en la **Figura 6**. Este diagrama reviste especial importancia dado que describe el proceso central por el cual los instrumentos de evaluación atraviesan durante su ciclo de vida completo, demostrando el comportamiento esperado del sistema en su función principal.

Etapas del Proceso de Validación

El flujo presenta tres etapas claramente delimitadas y secuenciales:

1. **Creación del Instrumento**: Iniciada por el docente responsable de la materia.
2. **Validación Departamental**: Ejecutada por el Director del Departamento correspondiente.
3. **Validación Institucional**: Realizada por la Secretaría Académica como instancia final.

Puntos de Decisión y Flujos Alternativos

El diagrama incorpora puntos de decisión estratégicos que contemplan tanto el flujo positivo (aprobación) como los flujos negativos (rechazo). Esta representación permite visualizar cómo reacciona el sistema ante el rechazo de un instrumento tipo A en cualquiera de las etapas de validación, incluyendo los mecanismos de retroalimentación y las rutas de retorno para corrección.

Procesos Secundarios

Los diagramas de flujo secundarios documentan funcionalidades complementarias del sistema, tales como:

Procesos de Gestión Administrativa:

- Gestión de usuarios y asignación de roles.
- Administración de departamentos, materias y cargos.
- Configuración de permisos para creación de instrumentos A.

Procesos de Comunicación, Notificación e ingreso al sistema:

- Sistema de mensajería bidireccional entre usuarios.
- Mecanismo de recuperación de contraseñas.
- Flujo de notificaciones automáticas del sistema.
- Registro de usuarios dentro del sistema.

Procesos de Consulta y Reportes:

- Consulta de historial de instrumentos A por departamento/materia.

- Generación de estadísticas del sistema.
- Búsqueda y consulta de docentes.
- Exportación de instrumentos A a formato PDF.

Procesos de Gestión Documental:

- Versionado de instrumentos A tras rechazo.
- Personalización de campos del instrumento A.
- Gestión de comentarios y retroalimentación.
- Cálculo automático de créditos docentes.

Análisis de seguridad y permisos

Una vez establecido el modelado del sistema con sus entidades, actores, funcionalidades y diagramas de flujo correspondientes, resulta fundamental analizar uno de los atributos de calidad más críticos en sistemas de información académica: la **seguridad del sistema**.

La estrategia de seguridad implementada se fundamenta en un modelo de **control de acceso basado en roles y permisos** (RBAC - Role-Based Access Control), el cual constituye un mecanismo esencial para garantizar la integridad, confidencialidad y disponibilidad de la información institucional.

Sistema de Roles Jerárquicos

El primer nivel de seguridad se establece mediante un **sistema de roles jerárquicos** que replica fielmente la estructura organizacional de la institución académica. Los roles implementados incluyen:

- **Docente**: Usuario operativo responsable de la creación y gestión de instrumentos tipo A.
- **Director de Departamento**: Supervisor académico con autoridad de validación departamental.
- **Observador**: Actor con permisos de consulta y retroalimentación sin capacidad de modificación.
- **Secretaría Académica**: Entidad administrativa con autoridad de validación final.
- **Administrador del Sistema**: Rol técnico con privilegios completos de configuración y gestión.

Cada rol posee un conjunto específico de **acciones permitidas** dentro del sistema, las cuales han sido detalladas en la sección 5.1.3.2 "Usos del Sistema", garantizando que cada tipo de usuario acceda únicamente a las funcionalidades correspondientes a sus responsabilidades institucionales.

Mecanismos de Autenticación Segura

El sistema implementa un **protocolo de activación de usuarios en dos etapas** que garantiza la seguridad desde el momento de la creación de cuentas:

Etapa 1 - Registro Administrativo: El administrador del sistema registra al nuevo usuario utilizando únicamente su dirección de correo electrónico institucional, estableciendo así un primer nivel de validación basado en la pertenencia a la organización.

Etapa 2 - Activación Segura: El usuario recibe automáticamente un correo electrónico conteniendo una contraseña temporal generada mediante algoritmos criptográficos seguros (generación aleatoria + SHA256). Al primer acceso al sistema, se requiere obligatoriamente el cambio de esta contraseña temporal por una elegida personalmente por el usuario.

Protección Criptográfica de Credenciales

Toda la información de autenticación se almacena en la base de datos utilizando **técnicas de cifrado irreversible** (hash), garantizando que ningún usuario, incluidos los administradores, tenga acceso a las contraseñas en texto plano. Esta implementación sigue las mejores prácticas de seguridad informática para la protección de credenciales.

Sistema de Permisos Granulares

El control de acceso se complementa con un **sistema de permisos granulares** inspirado en la filosofía de sistemas Unix, implementado mediante manipulación de bits que permite agregar o revocar permisos específicos de manera eficiente y flexible.

Permisos de Nivel Instrumento

Los permisos asociados a instrumentos tipo A determinan las **capacidades de interacción específicas** con cada documento:

- **Alcance**: Control sobre la capacidad de generar comentarios y observaciones en instrumentos específicos.
- **Granularidad**: Permite asignación selectiva de permisos de comentario por instrumento individual.
- **Referencia**: Especificaciones detalladas disponibles en Tabla 1.

Bit	Nombre	Descripción
1	Permiso de observador	Controlan si los observadores del sistema pueden crear comentarios sobre el instrumento A

2	Permiso de docente	Controlan si los docente asociados a la materia pueden crear comentarios sobre el instrumento A

Tabla 1. Permisos de Nivel Instrumento

Permisos de Nivel Cargo

Los permisos vinculados a cargos académicos poseen **mayor complejidad y alcance**, abarcando el control integral la materia y sus instrumentos asociados:

Operaciones Controladas:

- Consulta y edición de información de materias
- Creación, modificación y eliminación de instrumentos tipo A
- Gestión de equipos docentes y asignaciones
- Administración de contenidos programáticos

Alcance Departamental: Estos permisos se circunscriben al departamento académico correspondiente, garantizando la segregación de responsabilidades según la estructura organizacional institucional.

Referencia: Matriz completa de permisos disponible en Tabla 1.1

Bit	Nombre	Descripción
1	Permiso de consulta de materia	Controla que los docentes asociados a la materia puedan acceder a la información total de la misma.
2	Permiso de edición de materia	Controla que los docentes asociados a la materia puedan editar información de la misma.
3	Permiso de modificación de Instrumento A	Controla que los docentes asociados a la materia puedan modificar el Instrumento A en proceso de creación actual.
4	Permiso de creación/eliminación de Instrumento A	Controla que los docentes asociados a la materia puedan crear un instrumento A para comenzar con el proceso de validación o eliminar un Instrumento A en proceso de creación.

Tabla 1.1. Permisos de Nivel Cargo

Principios de Seguridad Implementados

La arquitectura de seguridad del sistema se fundamenta en los siguientes principios:

- **Principio de Menor Privilegio**: Cada usuario recibe únicamente los permisos mínimos necesarios para cumplir sus funciones.
- **Separación de Responsabilidades**: Las funciones críticas requieren la intervención de múltiples roles para su completitud.
- **Defensa en Profundidad**: Múltiples capas de seguridad (autenticación, autorización, permisos granulares).
- **Trazabilidad**: Registro completo de acciones para auditoría y seguimiento de seguridad.

Otros atributos de calidad

El análisis integral del proyecto revela un sistema que, más allá de la seguridad ya examinada, incorpora múltiples atributos de calidad esenciales para garantizar un software robusto, sostenible y adaptable a las necesidades institucionales a largo plazo.

Madurez Tecnológica

La arquitectura del sistema se fundamenta en tecnologías consolidadas y ampliamente adoptadas en la industria, garantizando estabilidad y confiabilidad operacional:

Backend: La implementación mediante **PHP con Laravel** proporciona un framework maduro con una extensa comunidad de desarrollo, documentación completa y un ecosistema robusto de librerías y herramientas de soporte.

Frontend: El uso de **Next.js con TypeScript** asegura un desarrollo moderno con tipado estático, optimizaciones automáticas de rendimiento y capacidades avanzadas de renderizado tanto del lado del servidor como del cliente.

Base de Datos: **MariaDB** ofrece estabilidad empresarial, compatibilidad con estándares SQL y capacidades de escalamiento horizontal y vertical probadas en entornos de producción críticos.

Mantenibilidad y Sostenibilidad

El sistema adopta principios de **diseño modular** que facilitan tanto el mantenimiento correctivo como evolutivo:

- **Separación de responsabilidades** entre capas de presentación, lógica de negocio y acceso a datos.

- **Implementación de patrones de diseño** reconocidos que simplifican la comprensión del código para futuros desarrolladores.
- **Documentación técnica integral** que incluye diagramas de arquitectura, especificaciones de API y guías de despliegue.

Buenas Prácticas de Desarrollo

La implementación sigue estándares reconocidos de la industria:

- **Versionado de código** mediante sistemas de control distribuido.
- **Testing automatizado** que incluye pruebas unitarias, de integración y funcionales.
- **Configuración basada en entornos** que facilita el despliegue en diferentes ambientes (desarrollo, testing, producción).

Escalabilidad Institucional

La arquitectura permite el crecimiento del sistema para atender demandas institucionales crecientes:

Capacidad de usuarios concurrentes: El sistema está diseñado para soportar el acceso simultáneo de múltiples usuarios sin degradación significativa del rendimiento, cumpliendo con los requerimientos no funcionales establecidos.

Distribución de carga: La separación entre frontend y backend mediante APIs REST facilita la implementación de estrategias de balanceado de carga y distribución geográfica de servicios.

Escalabilidad Vertical

Modularidad funcional: La estructura del sistema permite la incorporación de nuevos módulos y funcionalidades sin afectar componentes existentes, facilitando la evolución orgánica del software según necesidades emergentes.

Flexibilidad organizacional: El diseño contempla la adaptación a diferentes estructuras académicas, permitiendo su implementación en otras facultades o instituciones con variaciones en sus procesos administrativos.

Anexo III

Arquitectura del sistema

El sistema implementa una **arquitectura cliente-servidor basada en eventos** integrada con una **base de datos SQL** y un **Servidor de archivos frontend**. Esta decisión arquitectónica se

fundamenta en el principio de centralización de la información en entornos académicos seguros, donde la gestión, mantenimiento y operación del sistema permanecen bajo el control exclusivo de la institución académica correspondiente.

Arquitectura Cliente-Servidor con Eventos

El servidor principal implementa una **API REST** **monolítica modular**, combinando las ventajas de la simplicidad operacional de un monolito con la flexibilidad arquitectónica de la modularización. Esta aproximación permite mantener todos los servicios en una única aplicación desplegable, facilitando el desarrollo, testing y deployment, mientras conserva la separación lógica de responsabilidades mediante módulos bien definidos. También implementa un servidor **WebSocket** para la comunicación bidireccional entre el cliente y el servidor. El servidor secundario implementa un servicio de archivos frontend para el renderizado de la aplicación web en el navegador.

La arquitectura cliente-servidor con eventos se caracteriza por los siguientes componentes:

Servidor

- **Computadora central** que actúa como núcleo del sistema.
- **Receptor de peticiones** de múltiples usuarios simultáneos.
- **Procesador computacional** que resuelve las solicitudes utilizando recursos centralizados.
- **Generador de respuestas** que envía resultados procesados a los clientes.

Clientes

- **Usuarios finales** que acceden al sistema a través de navegadores web.
- **Interfaces de usuario** que generan peticiones al servidor.
- **Receptores de respuestas** que procesan y visualizan la información recibida.

Sistema de Eventos

- **Comunicación bidireccional** entre cliente y servidor.
- **Notificaciones en tiempo real** para actualizaciones inmediatas.
- **Sistema de mensajería** que permite interacciones asíncronas.
- **Gestión de estados** para mantener sincronización entre componentes.

Ventajas de la Arquitectura Cliente-Servidor con Eventos

Escalabilidad

- Capacidad de atender múltiples usuarios concurrentes.

- Distribución eficiente de la carga computacional.
- Posibilidad de expansión horizontal del servidor.

Mantenimiento Centralizado

- Actualizaciones y parches aplicados en un único punto.
- Reducción de costos de mantenimiento.
- Control centralizado de versiones y configuraciones.

Seguridad

- Control de acceso centralizado.
- Implementación uniforme de políticas de seguridad.
- Protección de datos en infraestructura institucional segura.

Comunicación en Tiempo Real

- Notificaciones instantáneas mediante eventos.
- Sincronización automática de datos entre usuarios.
- Mejora significativa en la experiencia de usuario.

Eficiencia de Recursos

- Procesamiento centralizado optimiza el uso de recursos computacionales.
- Reducción de redundancia de datos.
- Aprovechamiento de economías de escala.

Base de Datos SQL

La implementación de una base de datos SQL se justifica por la **existencia previa de una estructura formal de información** en el sistema, lo que facilitó la adaptación de la solución al problema específico del entorno académico.

Ventajas de la Base de Datos SQL

Integridad de Datos

- Constraints y reglas de integridad referencial.
- Transacciones ACID que garantizan consistencia.
- Prevención de inconsistencias en los datos.

Consultas Complejas

- Lenguaje SQL estándar para consultas avanzadas.

- Capacidades de análisis y reporting integradas.
- Joins eficientes para relacionar información.

Madurez Tecnológica

- Tecnología probada y estable.
- Amplia documentación y soporte comunitario.
- Herramientas de administración consolidadas.

Compatibilidad

- Estándares industriales ampliamente adoptados.
- Facilidad de integración con sistemas existentes.
- Portabilidad entre diferentes proveedores.

Beneficios para el Entorno Académico

La implementación de esta arquitectura en el contexto académico proporciona un **control institucional integral** que permite a la universidad mantener autonomía completa sobre la infraestructura tecnológica. Este control facilita el cumplimiento de normativas académicas específicas y garantiza la protección de datos sensibles dentro del perímetro institucional, aspecto fundamental en entornos educativos donde se maneja información confidencial de estudiantes, docentes e investigadores.

En términos de **continuidad operacional**, la arquitectura propuesta asegura la independencia de servicios externos, lo que resulta crítico para instituciones académicas que requieren disponibilidad constante del sistema según sus propias políticas y calendarios académicos. La capacidad de mantener respaldo y recuperación bajo control directo permite a la institución establecer sus propios estándares de disponibilidad y tiempo de recuperación, adaptándose a las necesidades específicas del entorno educativo.

La **flexibilidad y personalización** inherentes a esta arquitectura permiten la adaptación específica a las necesidades académicas particulares de cada institución. Esta característica resulta especialmente valiosa dado que los requerimientos educativos varían significativamente entre diferentes universidades, facultades y programas académicos. Además, la capacidad de modificación según requerimientos institucionales y la facilidad de integración con otros sistemas académicos existentes garantizan que la solución pueda evolucionar junto con las necesidades cambiantes de la institución, manteniendo la coherencia con el ecosistema tecnológico académico preestablecido.

Comunicación entre cliente y servidor.

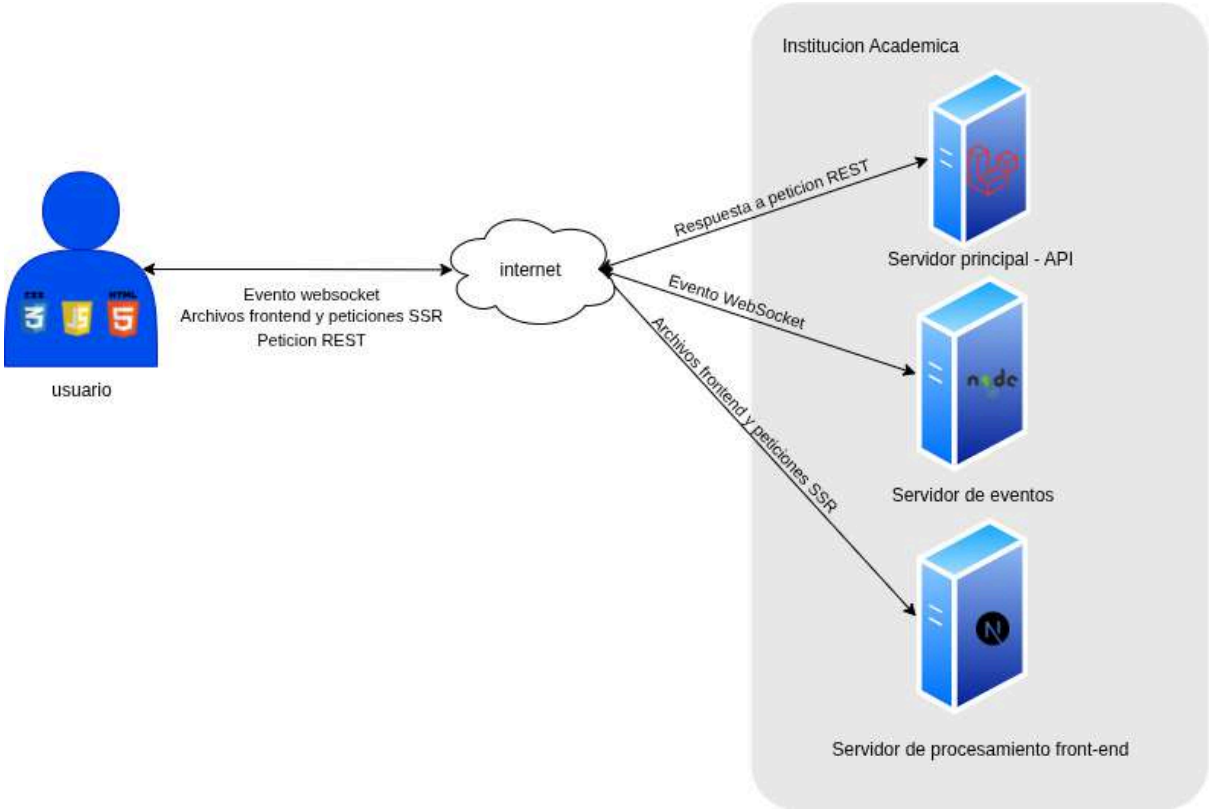


Figura 7. Diagrama de comunicación Cliente-Servidor

Como podemos observar en la Figura 7, el servidor principal implementa una **API REST** implementada en PHP - Laravel que establece las pautas de comunicación estandarizadas entre el usuario y el servidor para operaciones síncronas. Paralelamente, la comunicación en tiempo real entre el cliente y el servidor de eventos se gestiona mediante **WebSockets** implementado en NodeJS - Express , proporcionando un canal bidireccional persistente para interacciones asíncronas.

Así mismo existe un servidor de **Archivos frontend estático** y **SSR** en **NextJS** el cual cumple la función de enviar los archivos necesarios al usuario para que pueda visualizar la página web como renderizar componentes claves para aligerar la carga del cliente.

API REST (Representational State Transfer)

Una **API REST** constituye un servidor que implementa una arquitectura de servicios web basada en el protocolo HTTP, caracterizada por su forma estandarizada de acceder y manipular recursos. Esta tecnología establece un lenguaje de comunicación uniforme entre el cliente y el servidor, garantizando que ambas partes interpretan de manera inequívoca las operaciones solicitadas y las respuestas proporcionadas.

La arquitectura interna de la misma está construida de forma monolítica modular utilizando un patrón de diseño en capas que combina la **simplicidad operacional** del monolito con la **organización estructural** de los sistemas modulares. Esta aproximación permite mantener una única unidad desplegable mientras se estructura internamente en módulos cohesivos y capas especializadas.

Características del Diseño Monolítico Modular

El **enfoque monolítico** proporciona simplicidad de despliegue, debugging centralizado y gestión de transacciones ACID sin la complejidad de sistemas distribuidos. La **modularidad interna** organiza el código en componentes independientes con responsabilidades específicas, facilitando el mantenimiento, testing y desarrollo paralelo por equipos especializados.

Estructura en Capas

El **patrón de capas** establece una separación horizontal de responsabilidades: la capa de presentación maneja las interfaces y controladores, la capa de lógica de negocio encapsula las reglas del dominio académico, la capa de acceso a datos abstrae la persistencia, y la capa de infraestructura gestiona aspectos transversales como seguridad y monitoreo.

Esta arquitectura híbrida ofrece **escalabilidad controlada** y **evolución gradual**, permitiendo que módulos específicos puedan extraerse como microservicios independientes cuando los requerimientos de escala lo justifiquen, manteniendo la flexibilidad arquitectónica a largo plazo.

Ventajas de la API REST

La implementación de API REST proporciona **interoperabilidad universal** al basarse en estándares web ampliamente adoptados, permitiendo que diferentes tipos de clientes accedan al sistema sin requerir protocolos especializados. La **escalabilidad horizontal** se ve favorecida por la naturaleza stateless de las comunicaciones, mientras que la **simplicidad de implementación** reduce significativamente los costos de desarrollo y mantenimiento. Adicionalmente, la **caché nature** de las respuestas HTTP mejora el rendimiento del sistema al reducir la carga en el servidor para recursos frecuentemente solicitados.

WebSockets

WebSockets representa un protocolo de comunicación que establece un canal bidireccional persistente entre el cliente y el servidor, superando las limitaciones del modelo tradicional request-response de HTTP. Esta tecnología permite la creación de una línea de comunicación segura y continua donde ambas partes pueden intercambiar datos de manera asíncrona y en tiempo real.

Ventajas de WebSockets

La implementación de WebSockets proporciona **comunicación en tiempo real** que resulta fundamental para aplicaciones académicas colaborativas donde múltiples usuarios requieren actualizaciones instantáneas. La **eficiencia en el uso de recursos** se evidencia en la reducción del overhead de red y la eliminación de polling innecesario. La **experiencia de usuario mejorada** se manifiesta en interfaces más responsivas y dinámicas, mientras que la **flexibilidad de implementación** permite el desarrollo de funcionalidades avanzadas como notificaciones push, colaboración en tiempo real y sincronización automática de datos.

Servidor NextJS

La implementación de un servidor Frontend constituye un framework de desarrollo construido sobre React que proporciona una solución integral para la creación de aplicaciones web modernas.

Ventajas del servidor NextJS

La implementación de Next.js proporciona **versatilidad de renderizado** mediante el soporte nativo para Static Site Generation (SSG), Server-Side Rendering (SSR) y Client-Side Rendering (CSR), permitiendo optimizar cada página según sus requerimientos específicos. La **optimización automática** incluye code splitting, lazy loading, y optimización de imágenes que mejoran significativamente los tiempos de carga.

SSR (Server-Side Rendering)

El contenido HTML se genera en el servidor antes de enviarlo al navegador. Cada vez que el usuario solicita una página, el servidor procesa la petición, ejecuta el código, obtiene los datos necesarios y devuelve una página HTML completamente renderizada.

CSR (Client-Side Rendering)

El servidor envía un HTML básico con JavaScript, y es el navegador quien se encarga de generar todo el contenido dinámicamente usando JavaScript en el cliente.

Sinergia Tecnológica

La combinación de **API REST** y **WebSockets** en el sistema crea una arquitectura híbrida que aprovecha las fortalezas de ambas tecnologías. Las operaciones CRUD (Create, Read, Update, Delete) estándar se manejan eficientemente a través de la API REST, proporcionando una interfaz clara y cacheable para la gestión de datos. Simultáneamente, WebSockets facilita la comunicación de eventos en tiempo real, notificaciones inmediatas y actualizaciones colaborativas.

Esta arquitectura integral proporciona **rendimiento escalable** al combinar la velocidad de carga del SSR con la interactividad en tiempo real de WebSockets y la eficiencia de las API REST. La **flexibilidad arquitectónica** permite adaptar el tipo de renderizado y comunicación según los requerimientos específicos de cada funcionalidad, mientras que la **mantenibilidad del código** se ve beneficiada por la cohesión del ecosistema Next.js que unifica frontend, backend y capacidades de renderizado en una solución tecnológica integral.

Diseño de módulos

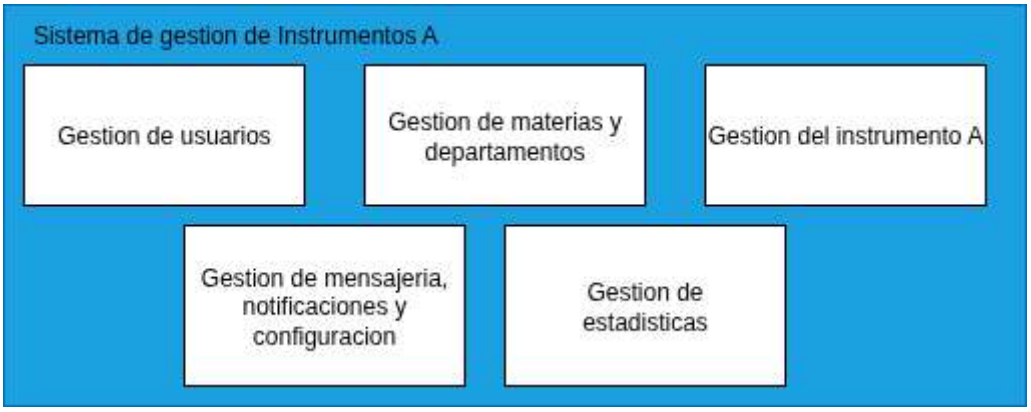


Figura 8. Diagrama de módulos

Como se ilustra en la Figura 8 (Diagrama de módulos), la arquitectura del sistema se estructura en cinco módulos principales: **Usuarios**, **Materias y Departamentos**, **Instrumentos**, **Mensajería, Notificaciones y Configuración**, y **Estadísticas**.

La decisión de segmentar el sistema en estos módulos específicos se fundamentó en la necesidad de establecer un alcance funcional claramente definido para cada componente, basándose en los requerimientos funcionales identificados. Esta separación modular proporcionó múltiples beneficios: facilitó la paralelización del desarrollo, estableció límites bien definidos entre componentes, y permitió abordar cada módulo como una unidad independiente. Consecuentemente, esta aproximación optimizó los procesos de diseño, desarrollo y testing del sistema.

Módulo de Gestión de Usuarios

El módulo de **Gestión de Usuarios** constituye el núcleo fundamental del sistema, encargándose de administrar todos los aspectos relacionados con la identidad, autenticación y autorización de los usuarios. Este componente centraliza las operaciones críticas que garantizan tanto la seguridad del acceso al sistema como la correcta gestión de perfiles y permisos.

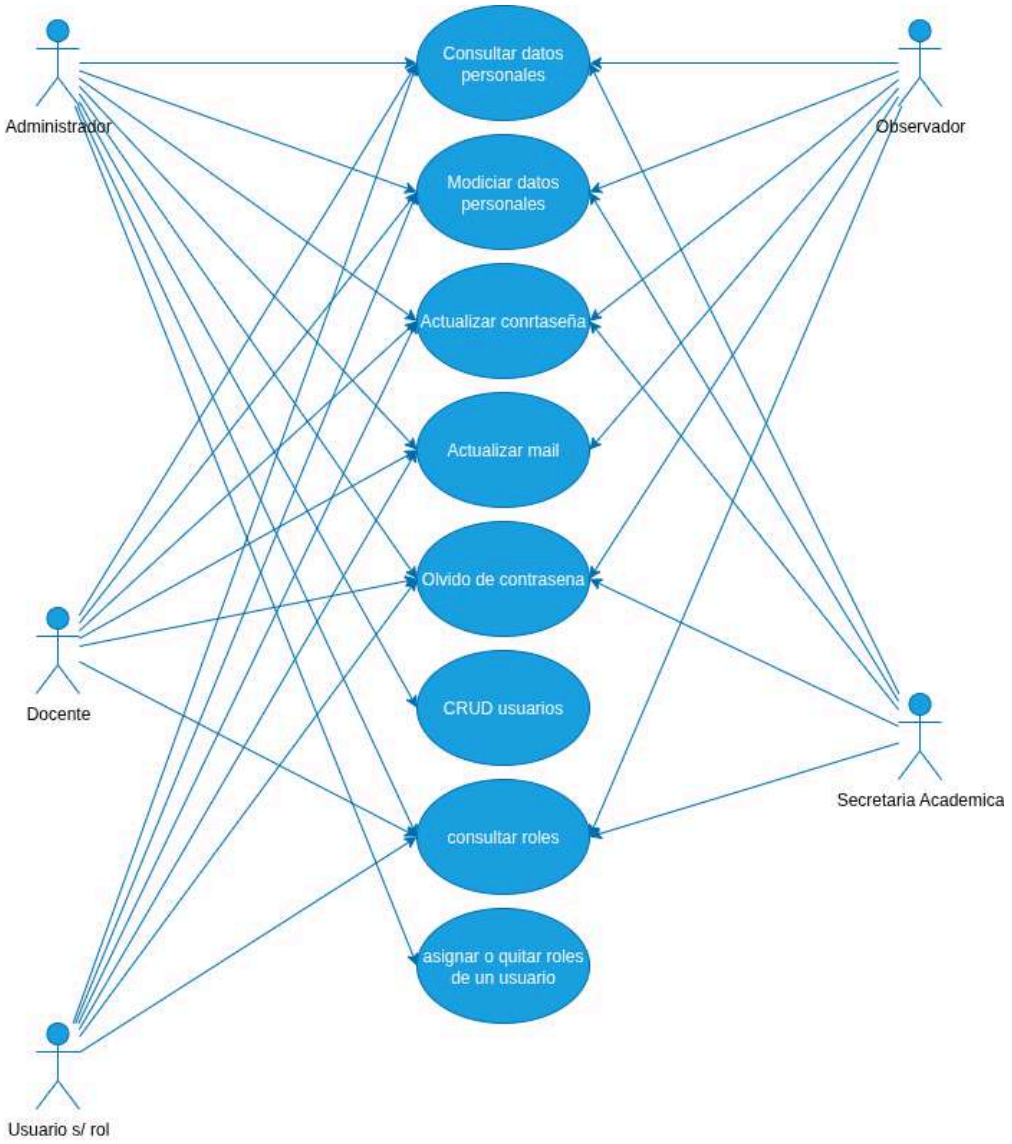


Figura 8.1. Diagrama de Casos de Uso para módulo de usuarios.

La Figura 8.1. ilustra los actores que interactúan directamente con el componente de gestión de usuarios, evidenciando dos categorías principales de funcionalidades. Por un lado, se identifican las operaciones administrativas, que incluyen el registro de nuevos usuarios, la asignación de roles y perfiles, y la consulta de información del sistema. Por otro lado, se observan las funcionalidades destinadas a usuarios generales, tales como el acceso y la salida del sistema, actualización de contraseña y consulta de datos personales.

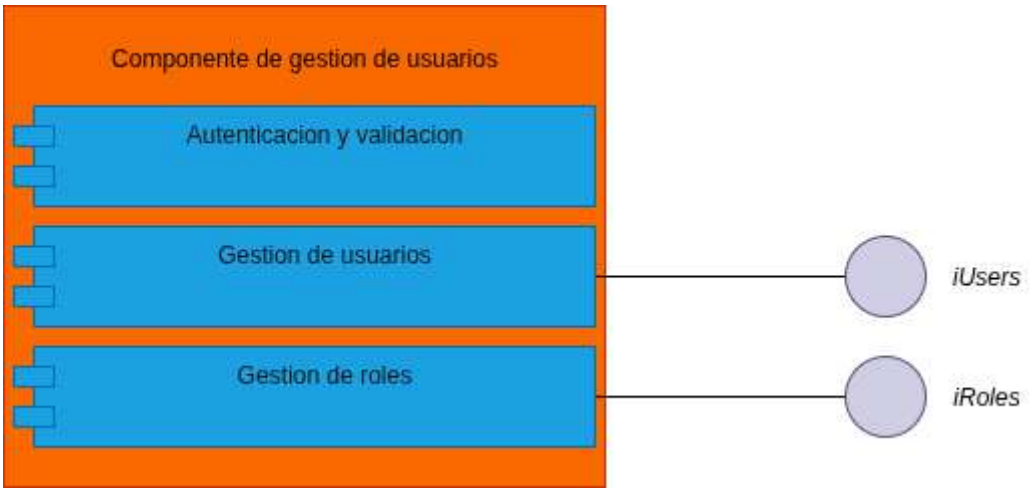


Figura 8.1.1. Componente de Gestion de Usuarios

El componente ilustrado en la Figura 8.1.1 presenta una estructura modular que organiza sus funcionalidades en sub-componentes especializados en el ingreso al sistema, la gestión integral de usuarios y roles. Esta división arquitectónica permitió un manejo atómico de los recursos del sistema, garantizando la coherencia e integridad de las operaciones. Asimismo, cada sub-componente expone una interfaz única y bien definida que establece protocolos de comunicación estandarizados, facilitando la interacción controlada y predecible entre los diferentes módulos del sistema.

Componentes del módulo

Autenticación y Validación: Sub-módulo responsable del control de acceso al sistema y la verificación de identidad de los usuarios autorizados. Este componente gestiona el ciclo completo de autenticación, desde el ingreso hasta el cierre de sesión, garantizando la seguridad en el acceso. Las funcionalidades principales incluyen:

- Autenticación de usuarios (login)
- Cierre de sesión de usuario (logout)
- Almacenamiento seguro de credenciales de acceso
- Recuperación de contraseñas
- Validación de credenciales de usuario
- Envio de mail de recuperación de contraseña

Gestión de Usuarios: Sub-módulo dedicado a la administración integral de la entidad usuario, orientado principalmente a operaciones administrativas que permiten el mantenimiento y consulta de información de usuarios del sistema. Las funciones centrales comprenden:

- Operaciones CRUD (Altas, Bajas, Modificaciones y Consultas) por usuario
- Creación de usuarios por parte de administradores y directores

- Modificación de datos personales no sensibles

Este sub-módulo expone la interfaz de comunicación "iUsers", que facilita la obtención de datos de usuarios por parte de otros componentes del sistema.

Gestión de Roles: Sub-módulo encargado de la administración del sistema de permisos y privilegios, responsable de asignar, remover y consultar los roles de usuario dentro del sistema. Las funcionalidades implementadas son:

- Consulta de roles del sistema
- Administración integral de roles del sistema

Adicionalmente, ofrece la interfaz "iRoles", que permite a otros módulos obtener, asignar o remover roles de usuarios específicos de manera controlada.

Módulo de Gestión de Materias y Departamentos

El módulo de **Gestión de Materias y Departamentos** se encarga de administrar la estructura académica e institucional del sistema educativo. Este componente es responsable de mantener la organización curricular y la jerarquía administrativa, estableciendo las relaciones entre las materias académicas, los departamentos institucionales y la asignación de responsabilidades docentes.

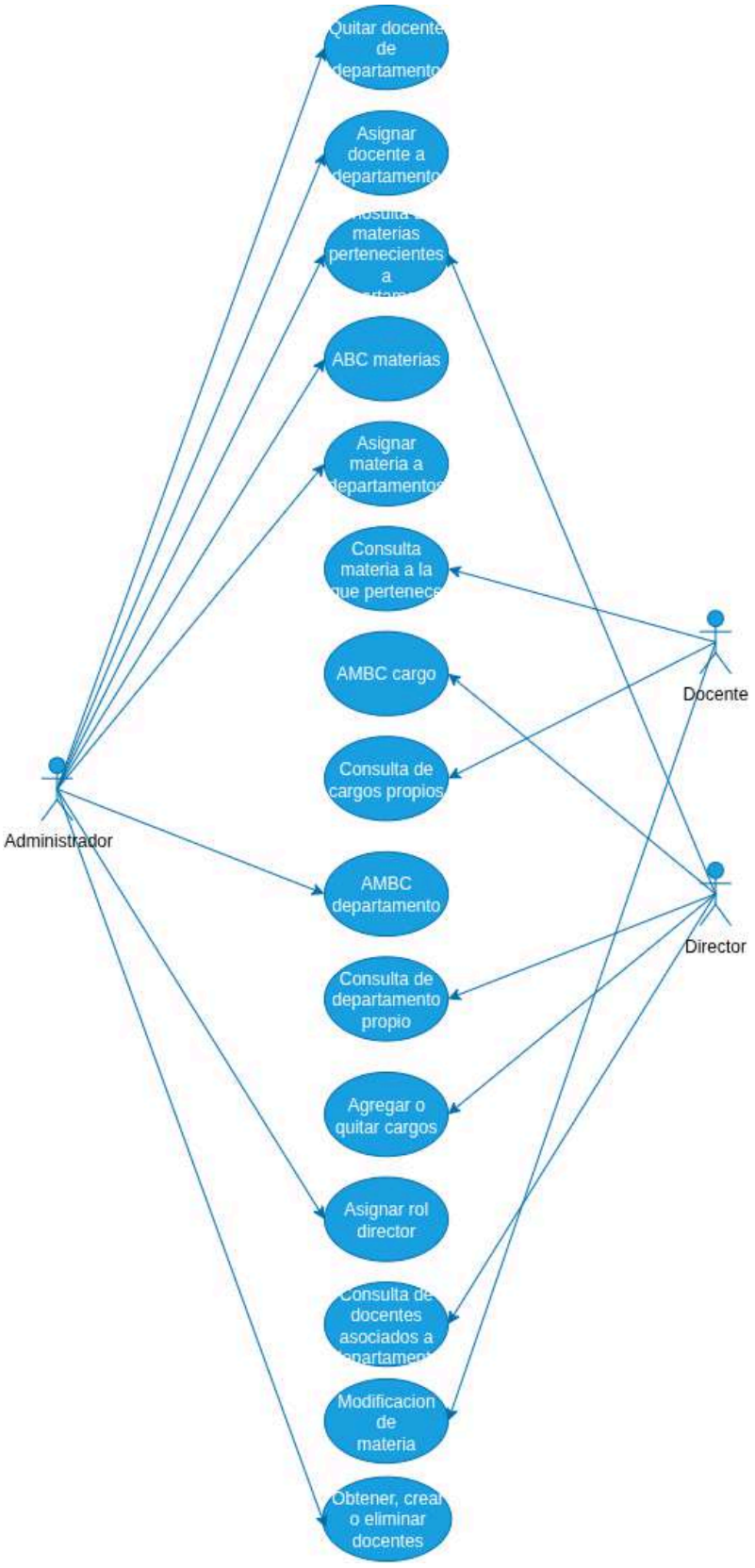


Figura 8.2. Diagrama de Casos de Uso para módulo de materias y departamentos.

En la Figura 8.2 se observa que los actores involucrados en este módulo corresponden a docentes, directores y administradores, lo cual refleja la naturaleza eminentemente administrativa de sus funcionalidades. Esta restricción de acceso se fundamenta en que las operaciones gestionadas por el módulo requieren privilegios elevados y están orientadas exclusivamente a tareas de administración del sistema.

Figura 8.2.1. Componente de Gestión de Materias y Departamentos.

El componente exhibido en la Figura 8.2.1 mantiene los principios de atomicidad en la gestión de recursos, estableciendo límites claros y bien definidos respecto al alcance funcional de cada sub-componente.

Componentes del Módulo

Gestión de Departamentos: Sub-módulo responsable de la administración integral de los departamentos académicos y su estructura organizacional. Sus funcionalidades principales incluyen:

- CRUD de departamentos
- CRUD de directores de departamento

Este componente expone la interfaz "iDepartments", que proporciona acceso controlado a la información departamental, incluyendo datos de directores y estructuras organizacionales.

Gestión de Materias: Sub-módulo dedicado a la administración académica de materias y sus relaciones curriculares. Las funcionalidades implementadas comprenden:

- CRUD para materias académicas

- Validación de permisos entre cargos y materias

El sub-módulo expone la interfaz "iSubject", cuya funcionalidad consiste en proporcionar información completa de las materias, incluyendo datos asociados como docentes asignados y departamento de pertenencia. Adicionalmente, valida si un docente posee los cargos y permisos necesarios para realizar acciones específicas sobre una materia determinada.

Gestión de Cargos: Sub-módulo intermediario que establece y administra las relaciones entre docentes, departamentos y materias mediante la gestión de cargos institucionales. Sus funciones centrales incluyen:

- CRUD de cargos institucionales
- Asignación y desasignación de cargos a docentes
- Implementación de la lógica de créditos académicos

La interfaz "iPositions" facilita el acceso a consultas sobre cargos asociados a departamentos específicos, incluyendo la verificación de asignaciones de cargos a docentes particulares.

Gestión de Docentes: Sub-módulo orientado a la administración integral del personal docente dentro del sistema. Su funcionalidad principal comprende:

- CRUD de docentes

La interfaz "iProfessors" proporciona acceso a la consulta de usuarios con perfil docente, facilitando la integración con otros módulos del sistema.

Módulo de Gestión de Mensajería, Notificaciones y Configuración

El **Módulo de Gestión de Mensajería, Notificaciones y Configuración** constituye el componente responsable de administrar las comunicaciones entre usuarios, gestionar el sistema de notificaciones del sistema y establecer la conectividad con el servidor WebSocket para comunicación en tiempo real. Adicionalmente, este módulo se encarga de la administración integral de la configuración del sistema y la gestión eficiente de su caché, garantizando un rendimiento óptimo en el acceso a los parámetros de configuración.

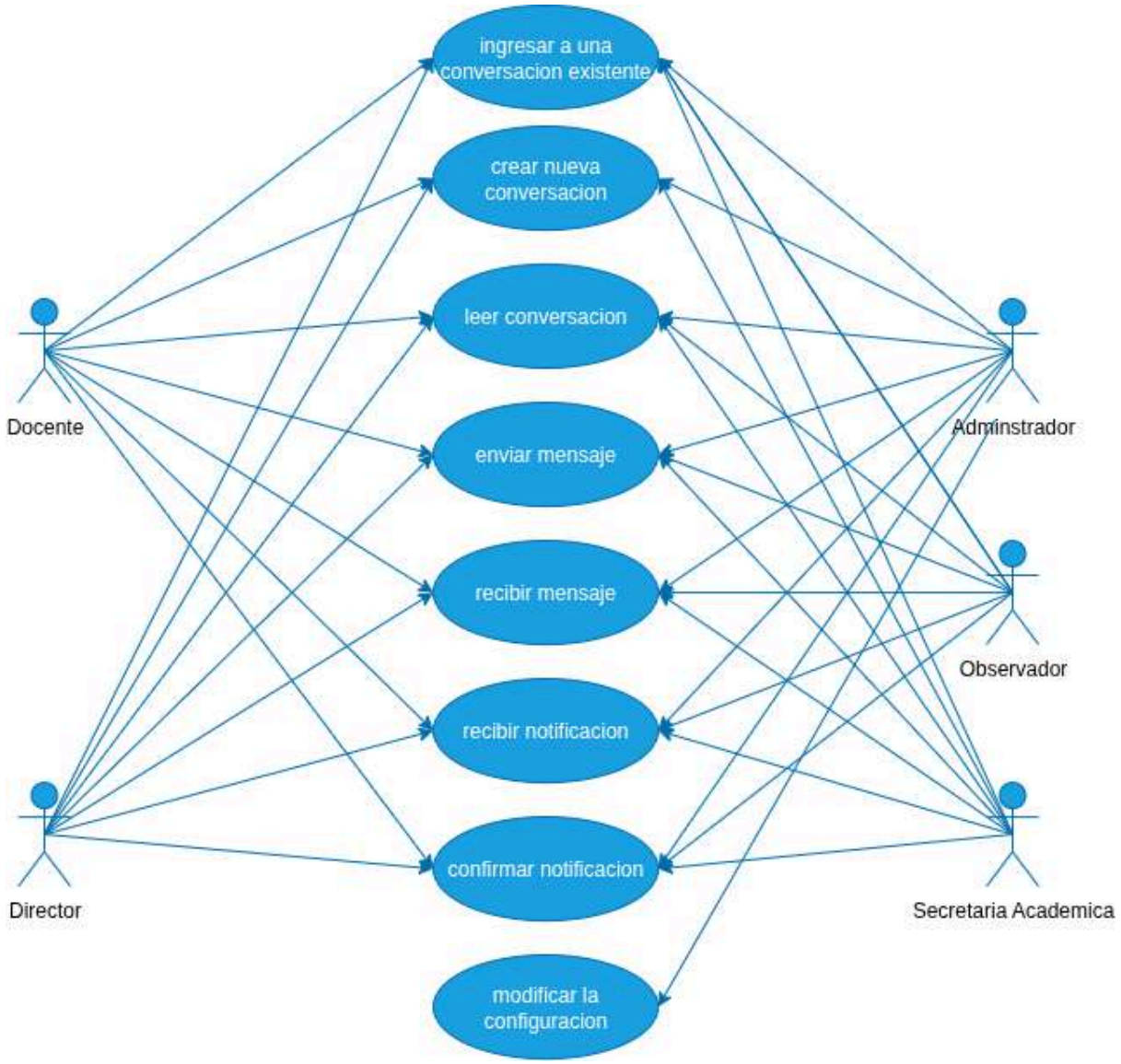


Figura 8.3. Diagrama de Casos de Uso para módulo de Mensajería, Notificaciones y Configuración.

En la Figura 8.3 se observa que todos los actores del sistema participan en las funcionalidades del módulo, dado que tanto las notificaciones como la mensajería constituyen servicios universales accesibles para todos los usuarios. No obstante, existe una diferenciación en los niveles de acceso, ya que la gestión administrativa de la configuración del sistema permanece restringida exclusivamente a los administradores, quienes poseen los privilegios necesarios para modificar los parámetros operacionales del sistema.

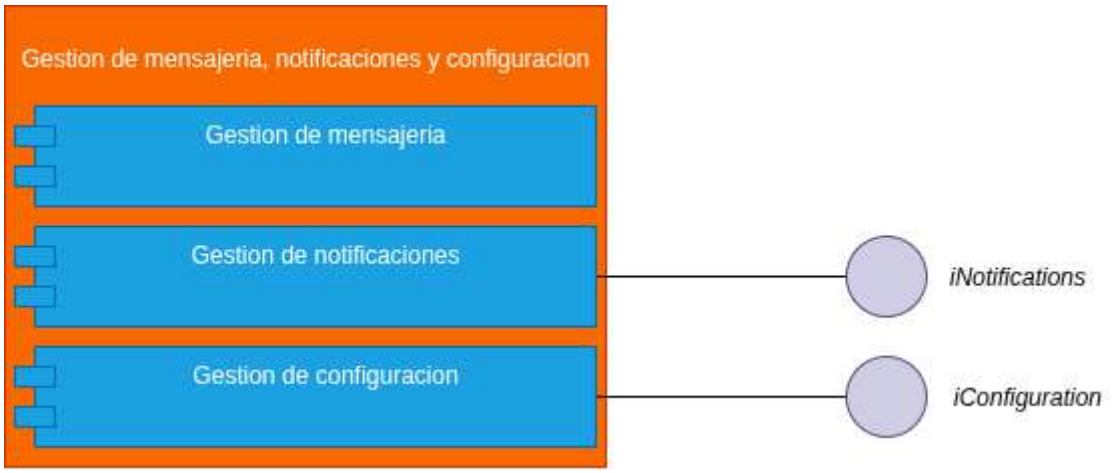


Figura 8.3.1. Componente de Mensajería, Notificaciones y Configuración.

Como el resto de componentes, la Figura 8.3.1 demuestra la atomicidad de los recursos, organizándose en módulos desacoplados que mantienen responsabilidades específicas y bien delimitadas.

Componentes del Módulo

Gestión de Mensajería: Sub-módulo responsable de la administración integral de mensajes y conversaciones entre usuarios del sistema. Sus funcionalidades principales incluyen:

- Creación de conversaciones
- Envío y recepción de mensajes internos
- Notificaciones de mensajería

Gestión de Notificaciones: Sub-módulo encargado de administrar el sistema de notificaciones de usuarios y establecer la comunicación con el servidor WebSocket para transmisión en tiempo real. Las funcionalidades implementadas comprenden:

- Creación de notificaciones
- Envío de notificaciones
- Eliminación de notificaciones

Este sub-módulo expone la interfaz "iNotifications", que permite a otros componentes del sistema generar notificaciones y distribuirlas a los usuarios correspondientes.

Gestión de Configuración: Sub-módulo dedicado a la administración de la configuración del sistema y el mantenimiento de su consistencia en memoria caché. Sus funcionalidades centrales incluyen:

- Modificación de configuración
- Consulta de configuración
- Administración de configuración en caché

Adicionalmente, expone la interfaz "iConfiguration", que facilita a los demás componentes del sistema el acceso controlado a los parámetros de configuración establecidos.

Módulo de Gestión de Estadísticas

El **Módulo de Estadísticas** constituye el componente responsable de generar y facilitar las consultas analíticas del sistema, proporcionando información consolidada sobre el uso y rendimiento de las diferentes funcionalidades.

Asimismo, las estadísticas al igual que la configuración se diseñaron para ser almacenadas en una memoria caché, con un tiempo de vida determinado, pasado eso, se vuelven a cargar en la memoria caché.

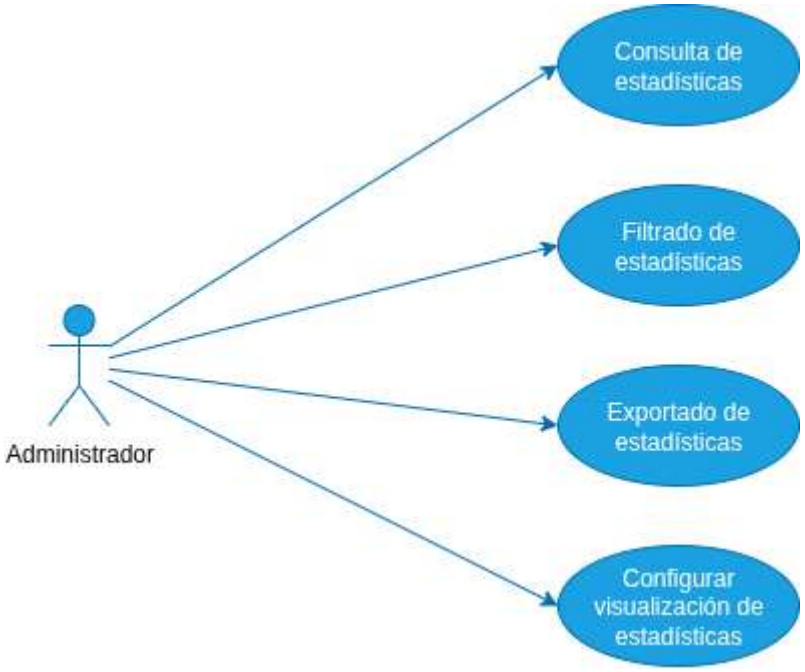


Figura 8.4. Diagrama de Casos de Uso para Modulo de Estadisticas.

En la Figura 8.4 se observa que el único actor involucrado en este módulo es el administrador, quien posee los privilegios exclusivos para realizar consultas y exportar estadísticas del sistema. Esta restricción de acceso se fundamenta en la naturaleza sensible de la información analítica y la necesidad de mantener el control sobre la divulgación de datos operacionales.

Figura 8.4.1 Componente de Estadísticas.

La Figura 8.4.1 presenta el componente de estadísticas, el cual exhibe una arquitectura simplificada que centraliza toda la funcionalidad del módulo en una estructura unificada. Esta decisión de diseño refleja la naturaleza específica y cohesiva de las operaciones estadísticas, que no requieren la complejidad modular presente en otros componentes del sistema.

Dentro de las estadísticas podemos observar los siguientes datos:

- Cantidad de usuarios en cada rol
- Cantidad de departamentos
- Cantidad de docentes en cada departamento
- Instrumentos A **(ciclo actual)**:
    - Aprobados
    - Pendientes por secretaría académica
    - Pendientes por departamento (se puede tener un valor total y un listado de los pendientes en cada departamento)
    - Creados pero no entregados a departamento (se puede tener un valor total y un listado de los pendientes en cada departamento)
    - Totales

Diseño del módulo de instrumento A.

El componente central del sistema se encarga de la administración del Instrumento A. Durante su diseño se prestó especial atención al acoplamiento con el resto del sistema, considerando su rol fundamental dentro de la arquitectura.

El Instrumento A

Estados

Los instrumentos A poseen un diagrama de estados que describe la situación en la que se encuentra cada instrumento en un momento determinado.

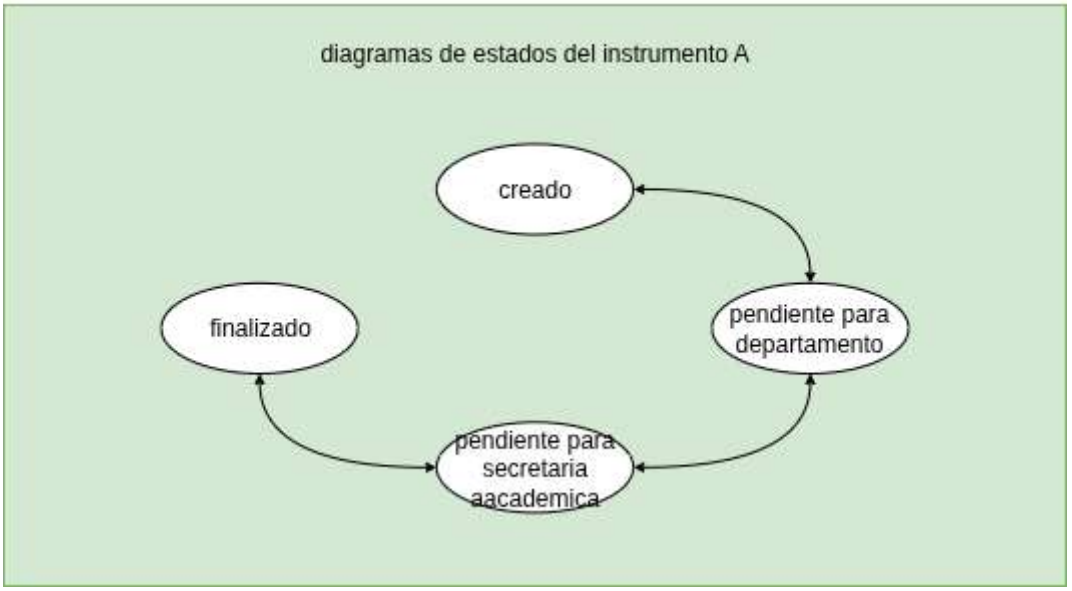


Figura 9. Diagrama de Estados del Instrumento A.

Como se observa en la Figura 9, cada Instrumento puede retroceder a su estado anterior, siguiendo siempre un flujo establecido que inicia en "Creado", continúa a través de dos validaciones y finaliza en "Finalizado".

Cada estado posee un ***scope*** diferenciado según el dominio de responsabilidad:

- **Estado de Creación**: Docentes de la Materia
- **Primera Validación**: Director del Departamento
- **Segunda Validación**: Secretaría Académica

Creación

El sistema proporciona opciones de autocompletado para simplificar el proceso de creación del Instrumento A, evitando que se convierta en una tarea tediosa. Las opciones disponibles son:

- Creación con información de la materia asociada
- Creación con información del instrumento del período anterior
- Creación con información de un instrumento específico
- Creación sin datos previos

Campos adicionales

El sistema permite agregar **campos extra** al instrumento, proporcionando flexibilidad para adaptarse a cualquier tipo de materia. Estas características incluyen:

- Los campos extra pueden ser creados y eliminados por el docente responsable del instrumento
- Posibilidad de habilitar **comentarios** por parte de observadores y otros docentes de la materia
- Los comentarios facilitan la comunicación entre los diferentes estados del instrumento
- Proporcionan retroalimentación inmediata y agilizan el proceso de creación

Restricciones

Las siguientes restricciones se aplican durante la creación y gestión del Instrumento A:

1. **Unicidad por materia**: No puede existir más de un Instrumento A en estado de creación por materia
2. **Períodos de creación**: Los instrumentos A solo pueden crearse dentro de los períodos establecidos en la configuración del sistema
3. **Métodos de creación**: Existen cuatro modalidades de creación disponibles:
    - Autocompletado con información de la materia asociada
    - Autocompletado con el instrumento aprobado del período anterior
    - Autocompletado con un Instrumento A específico
    - Sin autocompletado
4. **Permisos de modificación**: Solo docentes con cargo y permisos apropiados pueden cambiar el estado del instrumento
5. **Eliminación**: Únicamente se pueden eliminar Instrumentos A que se encuentren en estado de creación
6. **Reversión**: Secretaría Académica puede retroceder la finalización del Instrumento A si se encuentra dentro del período "post mortem" configurado
7. **Limpieza automática**: Los comentarios del Instrumento A se eliminan automáticamente al finalizar el período "post mortem" si el instrumento está finalizado

Diagrama de componentes

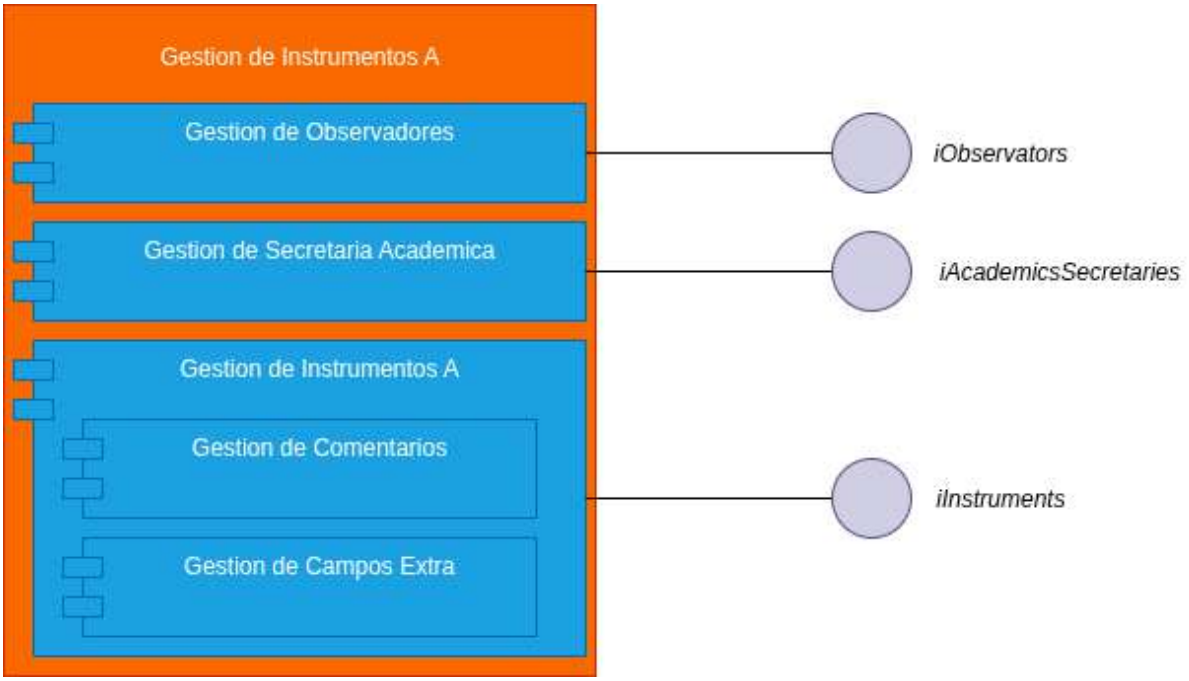


Figura 10. Diagrama de Componentes de Gestión de Instrumentos A.

Como se observa en la Figura 10, el módulo de Instrumentos A está compuesto por los siguientes elementos:

Gestión de Roles

Administra los últimos dos roles del sistema:

- **Observadores**: Funcionalidades de creación, eliminación y consulta
- **Secretaría Académica**: Funcionalidades de creación, eliminación y consulta

Asimismo presentan sus respectivas interfaces permitiendo al sistema hacer consultas y creaciones de usuarios con esos roles.

Submódulo de Gestión de Instrumentos A

Componente principal que maneja:

- Creación, modificación, eliminación y consulta de Instrumentos A
- Integración con módulos complementarios:
    - **Módulo de Campos Extra**: Permite creación, modificación, eliminación y consulta de campos personalizados
    - **Módulo de Comentarios**: Gestiona comentarios asociados a los instrumentos A con funcionalidades CRUD completas

Este submódulo proporciona una interfaz unificada para la consulta de Instrumentos A a lo largo de todo el sistema, facilitando la integración con otros componentes.

Diagrama de Casos de Uso

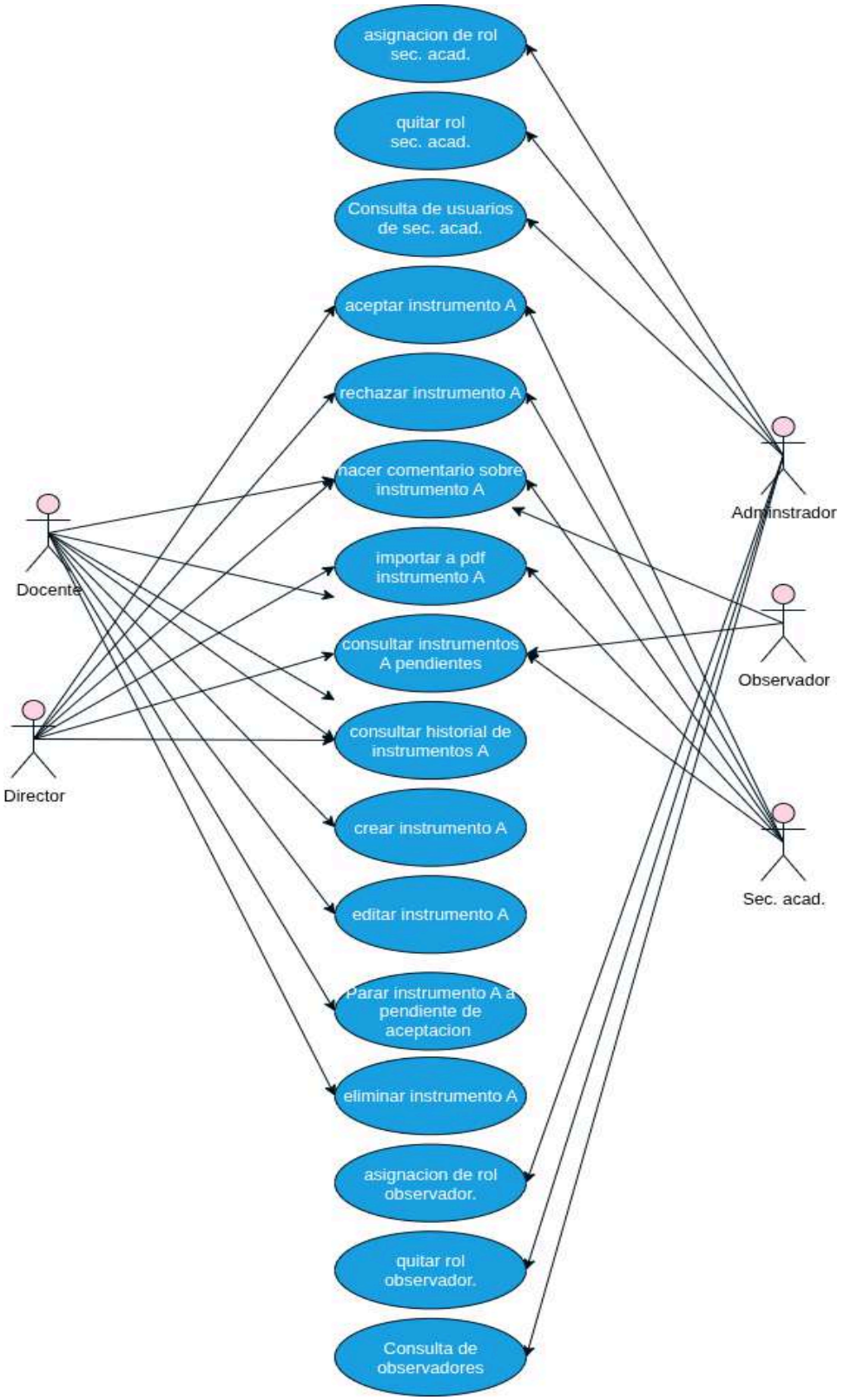


Figura 11. Diagrama de Casos de Uso de Módulo de Gestión de Instrumentos A

En la Figura 11 se pueden observar todas las funcionalidades descritas anteriormente implementadas en los diferentes componentes del sistema. Una característica destacable es que, al tratarse de un módulo central del sistema, todos los roles definidos en la aplicación están presentes y tienen interacciones específicas con el módulo de Instrumentos A.

Esta presencia integral de roles refleja la importancia estratégica del Instrumento A dentro del flujo de trabajo del sistema, ya que cada actor (Docentes, Directores de Departamento, Secretaría Académica y Observadores) desempeña funciones específicas en las diferentes etapas del ciclo de vida del instrumento, desde su creación hasta su finalización y posterior gestión.

Diagrama de componentes completo.

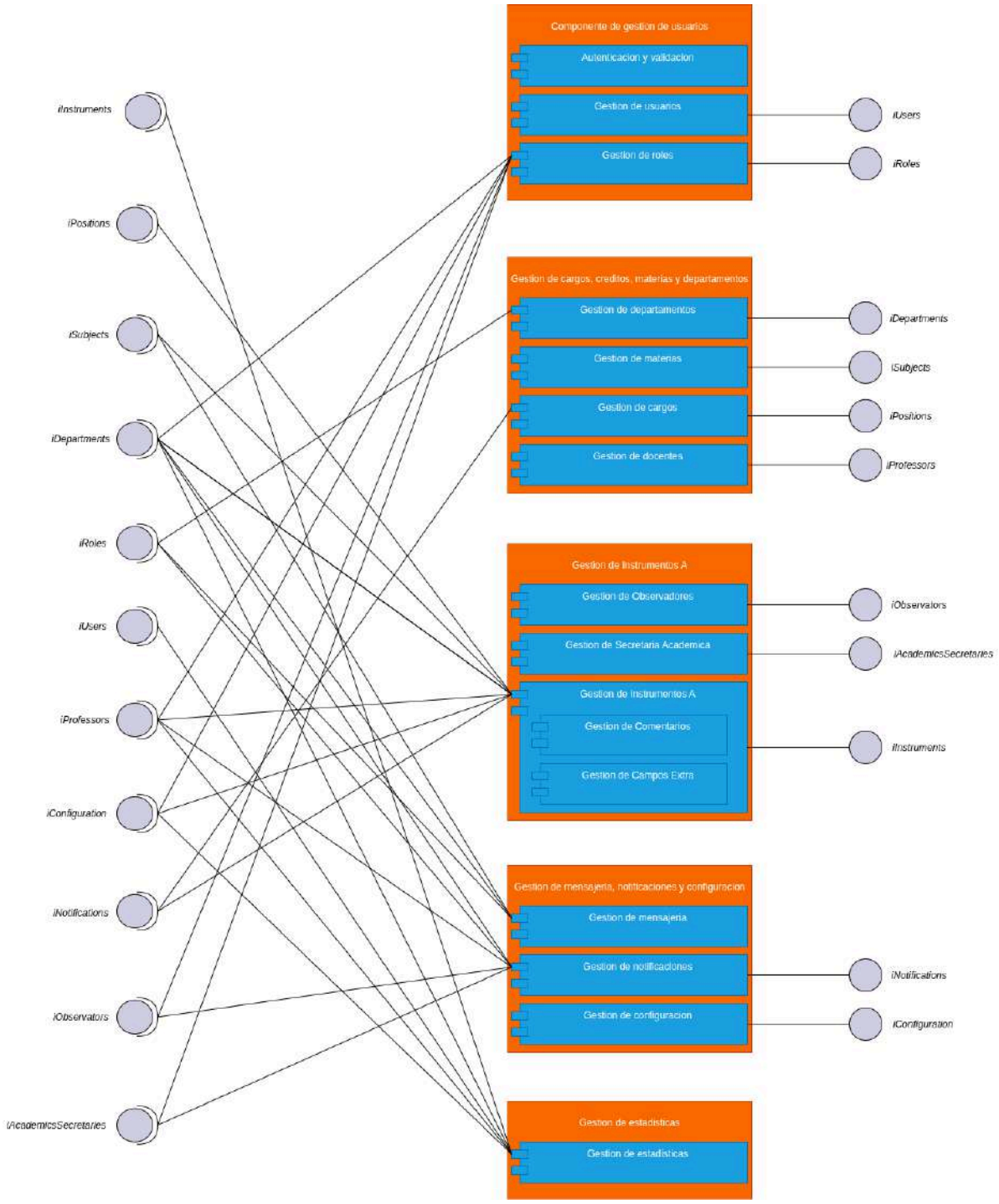


Figura 12. Diagrama de Componentes Completo

En la Figura 12 se presenta la arquitectura completa de componentes del sistema y sus interacciones. La disposición del diagrama ilustra claramente la separación entre:

- **Lado derecho**: Las interfaces que el sistema **consume** de componentes externos o dependencias
- **Lado izquierdo**: Las interfaces que el sistema **expone** para ser utilizadas por otros módulos o sistemas externos

Esta representación arquitectónica permite visualizar el flujo bidireccional de información y la forma en que el módulo de Gestión de Instrumentos A se integra con el ecosistema general del sistema. La clara diferenciación entre interfaces consumidas y expuestas facilita la comprensión de las dependencias del módulo y los servicios que proporciona, lo cual es fundamental para el mantenimiento, escalabilidad y futuras integraciones del sistema.

Tecnologías

Como se mencionó anteriormente, la tecnología principal para el desarrollo del sistema fue el framework **Laravel** implementado con el lenguaje de programación **PHP**. Si bien este framework ofrece capacidades de desarrollo fullstack, la decisión arquitectónica fue utilizar exclusivamente sus funcionalidades backend para el desarrollo de la API.

Adicionalmente, el backend incorpora un servidor WebSocket que facilita la comunicación bidireccional en tiempo real entre la API y el frontend.

Tecnologías Backend

Laravel Framework de desarrollo principal del backend del proyecto. Proporciona un ecosistema robusto para el desarrollo de aplicaciones web modernas y completas, respaldado por su madurez y amplia documentación. Su arquitectura MVC y conjunto de herramientas integradas facilitan el desarrollo ágil y mantenible.

Schedule (Cron Jobs) Para implementar las funcionalidades de programación automática de tareas del sistema, se utilizó **Schedule**, una biblioteca nativa de Laravel que permite la programación y ejecución automatizada de tareas según intervalos definidos.

Bcrypt Biblioteca de encriptación que implementa algoritmos de *hashing* seguros como SHA-256. Garantiza el almacenamiento seguro de contraseñas mediante técnicas de encriptación unidireccional, asegurando que las credenciales permanezcan inaccesibles incluso para administradores del sistema.

MariaDB como sistema gestor de bases de datos, seleccionado por su rendimiento, rapidez y capacidad de escalabilidad, características que lo hacen idóneo para la solución planteada.

Tecnologias Front End

Next.js: Framework de React que se utiliza para construir aplicaciones web de alto rendimiento. Añade características y herramientas que facilitan el desarrollo de aplicaciones modernas y optimizadas para SEO, como el enrutamiento, la generación estática (SSG) y el renderizado del lado del servidor (SSR), ofreciendo una experiencia de desarrollo más completa que solo usar React.

Tecnologías compartidas

JWT (JSON Web Token) Dado el diseño *stateless* de la API, se requería un mecanismo para mantener la identificación de usuarios posterior a la autenticación. Los JWT proporcionaron una solución que no solo permite validar la identidad del usuario, sino también encapsular información relevante del usuario en cada petición, eliminando la necesidad de almacenar sesiones en el servidor. Se utiliza tanto en el backend para la generación y validación de tokens, como en el frontend para el almacenamiento seguro y envío de credenciales.

Node.js y Express.js Runtime de JavaScript y framework web minimalista utilizados para implementar el servidor WebSocket independiente. Node.js proporciona eficiencia en el manejo de operaciones asíncronas y comunicaciones en tiempo real, mientras que Express.js facilita la creación del servidor actualizado como mediador de comunicación entre el servidor principal de la API y los clientes.

Socket.IO Biblioteca especializada que permite la implementación de comunicación bidireccional y en tiempo real entre el cliente y el servidor. En el sistema, gestiona el registro de usuarios conectados, permite el envío selectivo de eventos según roles y usuarios específicos, y proporciona un canal de comunicación asíncrono para notificaciones y actualizaciones en tiempo real. Se utiliza tanto en el servidor WebSocket como en el cliente frontend.

Diseño del despliegue del sistema

El despliegue desempeña un rol fundamental en el sistema, dado que representa la instancia activa y operativa del software en un entorno de producción. Por este motivo, un diseño adecuado de la distribución del sistema en el servidor resulta esencial, no sólo para facilitar su comprensión y mantenimiento, sino también para garantizar un rendimiento óptimo, escalabilidad y alta disponibilidad.

Estrategia de Despliegue

El enfoque principal del despliegue se basó en la contenerización de los servicios y su intercomunicación mediante arquitectura de microservicios. Esta aproximación permite:

- **Aislamiento de servicios**: Cada componente opera de manera independiente, reduciendo el acoplamiento y facilitando el mantenimiento.
- **Escalabilidad horizontal**: Los servicios pueden replicarse según la demanda sin afectar al resto del sistema.
- **Portabilidad**: La infraestructura puede desplegarse en diferentes entornos de manera consistente.
- **Gestión simplificada**: Las actualizaciones y el versionado de servicios se realizan de forma independiente.

Como tecnología de contenerización principal se utilizó **Docker**, debido a su madurez tecnológica, amplio soporte comunitario y la familiaridad de los estudiantes con esta herramienta. Docker proporciona las siguientes ventajas:

- Entornos de ejecución ligeros y eficientes
- Reproducibilidad del entorno de desarrollo en producción
- Integración con herramientas de orquestación (Docker Compose, Kubernetes)
- Gestión simplificada de dependencias

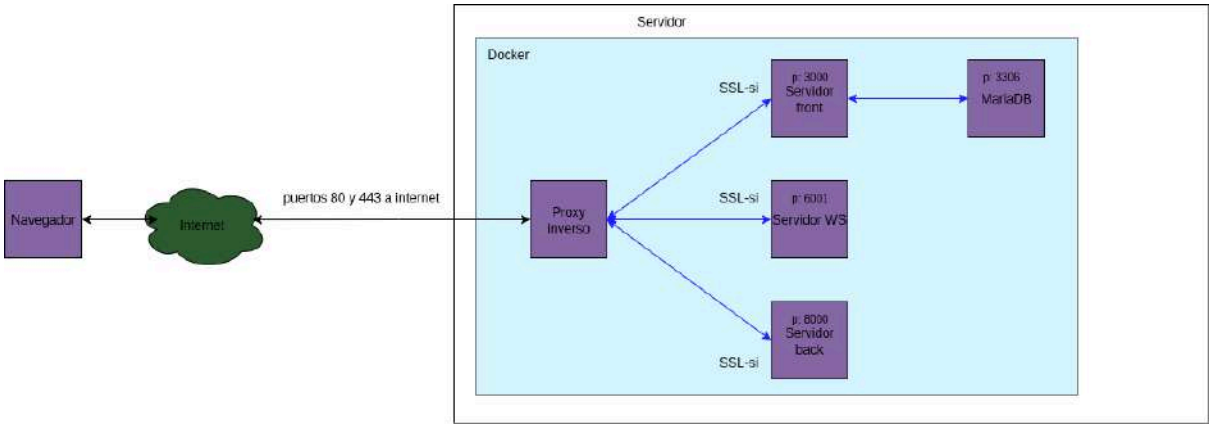


Figura 13. Diagrama de Despliegue

En la Figura 13 se presenta el Diagrama de Despliegue del sistema. Como puede observarse, la arquitectura está compuesta por los siguientes elementos:

Contenedores de Servicios

Cada servicio se encuentra contenerizado en una unidad independiente que incluye:

- **Aplicación**: Código fuente y dependencias necesarias
- **Runtime**: Entorno de ejecución específico (Nginx, NextJS, Laravel y Node)
- **Configuración**: Variables de entorno y archivos de configuración

Red Docker

Los servicios se comunican a través de una red Docker privada. Las flechas azules en el diagrama representan la comunicación bidireccional interna dentro de esta red, garantizando:

- Aislamiento de la red pública
- Resolución de nombres mediante DNS interno
- Comunicación eficiente entre contenedores

Proxy Inverso

El proxy actúa como punto de entrada único al sistema y posee comunicación abierta hacia Internet. Sus funciones principales incluyen:

- Enrutamiento de solicitudes a los servicios correspondientes
- Balanceo de carga entre instancias de servicios
- Terminación SSL/TLS
- Protección contra ataques comunes (DDoS, inyección, etc.)

Certificados SSL/TLS

Cada servicio cuenta con certificados SSL/TLS, lo que permite establecer una comunicación segura y cifrada desde el exterior de la red. Esta implementación garantiza:

- **Confidencialidad**: Los datos transmitidos están cifrados
- **Integridad**: Se verifica que los datos no han sido alterados
- **Autenticación**: Se valida la identidad de los servicios

Segregación de Puertos

Para facilitar la separación lógica dentro de la red Docker, cada servicio utiliza un único puerto. Si bien esta asignación no es estrictamente necesaria debido a que cada contenedor posee su propia dirección IP privada, esta decisión se tomó principalmente con los siguientes objetivos:

- Mejorar la comprensión de la configuración del sistema
- Facilitar la depuración y el monitoreo de servicios
- Simplificar la documentación de la infraestructura
- Permitir mapeos explícitos en el archivo de configuración

Anexo IV

Análisis de riesgos

Escalabilidad Institucional

Descripción: Uno de los riesgos críticos identificados es la escalabilidad del sistema hacia otras instituciones académicas. Este riesgo se fundamenta en la posible variabilidad de los procesos de gestión de Instrumentos A entre diferentes facultades o universidades, donde podrían existir actores adicionales, estados específicos o flujos de trabajo particulares que el sistema actual no contempla.

Impacto Potencial: Alto - Podría limitar significativamente la capacidad de expansión del sistema y requerir rediseños arquitectónicos costosos.

Medidas de Mitigación:

- Implementación de arquitectura modular basada en tecnologías modernas que soporten grandes volúmenes de usuarios concurrentes
- Adopción de estándares de desarrollo y buenas prácticas que faciliten el mantenimiento y extensión del sistema

Resistencia al Cambio Organizacional

Descripción: Posible resistencia de docentes y personal administrativo a abandonar procesos manuales establecidos, particularmente en instituciones con cultura organizacional más tradicional.

Impacto Potencial: Alto - Podría resultar en baja adopción del sistema y fracaso del proyecto.

Medidas de Mitigación:

- Programa de capacitación integral diferenciado por roles
- Implementación gradual con períodos de transición
- Comunicación clara de beneficios y mejoras en eficiencia

Dependencia Tecnológica

Descripción: Riesgo de interrupción de procesos académicos críticos en caso de fallas del sistema, especialmente durante períodos de alta demanda.

Impacto Potencial: Alto - Podría afectar directamente los cronogramas académicos institucionales.

Medidas de Mitigación:

- Implementación de sistemas de respaldo y recuperación ante desastres
- Mantenimiento de procedimientos manuales de emergencia documentados
- Monitoreo proactivo del sistema con alertas automáticas

Evolución de Requerimientos Institucionales

Descripción: Riesgo de cambios en normativas académicas o requerimientos institucionales durante el desarrollo o después de la implementación.

Impacto Potencial: Medio - Podría requerir modificaciones no planificadas del sistema.

Medidas de Mitigación:

- Diseño modular que facilite modificaciones futuras
- Documentación exhaustiva de decisiones de diseño
- Canal formal de gestión de cambios con stakeholders institucionales

Migración de Servidores y Distribución de Cargas

Descripción: Riesgo asociado a la necesidad de migrar el sistema entre diferentes entornos institucionales o de distribuir la carga de trabajo de manera eficiente cuando se requiera escalamiento horizontal. La dependencia de configuraciones específicas del servidor podría generar interrupciones de servicio o inconsistencias en el comportamiento del sistema durante procesos de migración o rebalanceo.

Impacto Potencial: Medio - Las migraciones fallidas podrían resultar en interrupciones prolongadas del servicio durante períodos académicos críticos, afectando directamente los procesos de acreditación.

Medidas de Mitigación:

- Implementación de contenedores *Docker* para garantizar portabilidad y consistencia entre entornos
- Uso de orquestadores de contenedores (*Kubernetes*) para automatizar el escalado y distribución de cargas
- Configuración de ambientes reproducibles que eliminen dependencias específicas del hardware o sistema operativo subyacente

12. Bibliografía

Anderson, D. (2010). *Kanban: Successful evolutionary change for your technology business*. Blue Hole Press.

bcrypt. (2025). *bcrypt - npm*. npm. https://www.npmjs.com/package/bcrypt

Laravel Documentation. (2025). Task scheduling - Laravel documentation. Laravel LLC. https://laravel.com/docs/scheduling

Docker Documentation. (2025). *Docker: Accelerated container application development*. Docker Inc. https://docs.docker.com

Express.js Documentation. (2025). *Express - Node.js web application framework*. OpenJS Foundation. https://expressjs.com

Git Documentation. (2025). *Git - Documentation*. Software Freedom Conservancy. https://git-scm.com/doc

GitHub. (2025). *GitHub: Let's build from here*. GitHub Inc. https://github.com

JSON Web Token. (2025). *JWT.io - JSON Web Tokens*. Auth0. https://jwt.io

Kubernetes Documentation. (2025). *Kubernetes documentation*. The Linux Foundation. https://kubernetes.io/docs

Laravel Documentation. (2025). *Laravel - The PHP framework for web artisans*. Laravel LLC. https://laravel.com/docs

Larman, C. (2004). *Applying UML and patterns: An introduction to object-oriented analysis and design and iterative development* (3rd ed.). Prentice Hall.

MariaDB Documentation. (2025). *MariaDB server documentation*. MariaDB Foundation. https://mariadb.org/documentation

MongoDB Documentation. (2025). *MongoDB manual*. MongoDB Inc. https://www.mongodb.com/docs

Next.js Documentation. (2025). *Next.js by Vercel - The React framework*. Vercel Inc. https://nextjs.org/docs

Node.js Documentation. (2025). *Node.js documentation*. OpenJS Foundation. https://nodejs.org/docs

Pest PHP. (2025). *Pest - The elegant PHP testing framework*. Nuno Maduro. https://pestphp.com

PHP Documentation. (2025). *PHP: Hypertext preprocessor*. The PHP Group. https://www.php.net/docs.php

Policies (Laravel). (2025). *Authorization - Laravel documentation*. Laravel LLC. https://laravel.com/docs/authorization#creating-policies

PostgreSQL Documentation. (2025). *PostgreSQL: The world's most advanced open source database*. PostgreSQL Global Development Group. https://www.postgresql.org/docs

Redis Documentation. (2025). *Redis documentation*. Redis Ltd. https://redis.io/docs

Tanenbaum, A. S., & van Steen, M. (2007). *Distributed systems: Principles and paradigms* (2nd ed.). Pearson Prentice Hall.

TypeScript Documentation. (2025). *TypeScript: JavaScript with syntax for types*. Microsoft Corporation. https://www.typescriptlang.org/docs

WebSocket Protocol. (2011). *RFC 6455 - The WebSocket protocol*. Internet Engineering Task Force (IETF). https://datatracker.ietf.org/doc/html/rfc6455