

Desarrollo Front-End de CoreSafe: Aplicación Web para la Planificación y Gestión de Programas de Registro Conductual (PRC) y la Visualización de Observaciones en Minería

Jeyson Alexandro Fernández Fabres

jeyson.fernandez@sansano.usm.cl

Catherine Gómez
Profesora Guía

David Larrondo
Profesor Correferente

Resumen: En el sector minero, la gestión manual de Programas de Registro Conductual (PRC) y observaciones en terreno genera ineficiencias operativas, errores de transcripción y demoras en la implementación de medidas preventivas. Este proyecto se centra en el desarrollo de CoreSafe, una plataforma web que digitaliza y optimiza estos procesos, mejorando la trazabilidad y el análisis de la información relacionada con la seguridad laboral. La solución integra módulos para la gestión de PRC, administración de usuarios, creación de plantillas y notificaciones automáticas, contruidos bajo una arquitectura modular y escalable con Vue.js. El desarrollo se llevó a cabo aplicando la metodología ágil Scrum, con iteraciones cortas, pruebas unitarias e integrales que aseguran la estabilidad y rendimiento del sistema. La propuesta permite centralizar los datos de seguridad en tiempo real, reducir errores humanos y fortalecer la eficiencia operativa de la empresa Indemin, contribuyendo a una cultura de seguridad más proactiva y a la continuidad productiva en la industria minera.

Palabras Clave: digitalización, seguridad laboral.



CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): ☒ Memoria o trabajo de título ☐ Tesis de Postgrado

Título del trabajo: Desarrollo Front-End de CoreSafe: Aplicación Web para la Planificación y Gestión de Programas de Registro Conductual (PRC) y la Visualización de Observaciones en Minería.

Nombre del candidato(a): Jeyson Alexandro Fernández Fabres

Carrera / Grado: Ingeniería en Informática

Campus: Viña del mar

Departamento: ELINF

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Catherine Gómez Barrera, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución.

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL (marcar una opción)

☒ El trabajo **NO contiene** información que amerite confidencialidad y puede ser publicado de inmediato en repositorio con acceso abierto.

☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (**embargo**) por (**marcar una opción**):

☐ 6 meses ☐ 12 meses ☐ 2 años ☐ 3 años ☐ 5 años ☐ 10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

4.- FIRMAS

Profesor(a) guía o director(a) de memoria o tesis:

Fecha: 16-01-2026

Firma:

Estudiante o Candidato(a):

Fecha: 16/01/2026

Firma:

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.

1 Introducción

1.1 Contexto y relevancia del desarrollo de componentes específicos en el proyecto.

En la actualidad, la era digital ha transformado radicalmente la manera en que las organizaciones gestionan sus procesos, dejando de ser una opción para convertirse en una condición indispensable para mantener la competitividad en los mercados globales.

Esta realidad es especialmente evidente en sectores de alta complejidad, como el industrial y el minero, donde la seguridad laboral y la prevención de riesgos son el eje fundamental para garantizar la continuidad productiva, y la protección de las personas representan factores críticos no solo para la continuidad operativa, sino también para el cumplimiento de normativas legales y la protección de la vida de los trabajadores. En este contexto, surge la necesidad de implementar una solución tecnológica que permita integrar la digitalización y la automatización como parte esencial de la gestión organizacional, promoviendo con ello mayor eficiencia y un enfoque preventivo en la administración de los riesgos asociados a la operación.

La empresa Indemin, reconocida como líder en logística interna y movimiento de carga, se encuentra actualmente en un proceso de transformación enfocado en fortalecer su competitividad mediante la digitalización de sus procesos. En coherencia con su estrategia de crecimiento, la organización ha priorizado la modernización de sus sistemas de control y la consolidación de una cultura orientada a la seguridad y a la mejora continua. Sin embargo, las metodologías tradicionales empleadas en la gestión de registros conductuales y observaciones en terreno evidencian limitaciones significativas que restringen la eficiencia en estas prácticas.

En la actualidad, los programas de registro conductual (PRC) y las observaciones de campo se realizan de manera manual, a través de formularios impresos que los observadores completan durante sus jornadas en las faenas. Este procedimiento no solo presenta un alto riesgo de errores en la escritura y en la interpretación de la información, sino que también genera una duplicidad de tareas, ya que posteriormente los documentos deben ser transcritos manualmente a plantillas de Excel. Este proceso, además de demandar un esfuerzo administrativo considerable, ralentiza la obtención de resultados y limita la capacidad de análisis oportuno. Como consecuencia, la detección de patrones conductuales se ve restringida, la generación de reportes carece de agilidad y la implementación de medidas preventivas se retrasa. Asimismo, en los casos en que se identifican conductas riesgosas, la vinculación de dichas observaciones con programas de capacitación específicos resulta poco eficiente, lo que dificulta la respuesta inmediata ante situaciones que podrían derivar en incidentes.

De esta manera, la problemática identificada radica en que la gestión manual de la información relacionada con la seguridad y el comportamiento de los trabajadores no solo representa un procedimiento engorroso y repetitivo, sino que también repercute negativamente en la productividad de la empresa y en su capacidad para diferenciarse en un mercado altamente competitivo. Frente a este escenario, se plantea la necesidad de diseñar e implementar una solución tecnológica que permita optimizar el registro, el análisis y la toma de decisiones en relación con la seguridad operacional, contribuyendo así a la consolidación del proceso de digitalización que la empresa Indemin ha definido como un eje estratégico de desarrollo.

1.2 Definición del problema.

El problema central que se busca abordar en este proyecto se relaciona con la ineficiencia y las limitaciones del sistema manual que actualmente utiliza la empresa Indemin para la gestión de registros conductuales (PRC) y observaciones en terreno. En la práctica, los observadores deben imprimir formularios, completarlos a mano durante las faenas y posteriormente trasladar esta información a oficina, donde otra persona transcribe los datos en plantillas Excel. Este procedimiento, además de ser repetitivo y demandar una alta carga administrativa, genera una serie de dificultades que afectan tanto a la productividad como la seguridad organizacional.

En primer lugar, el uso de registros manuales incrementa el riesgo de errores en la escritura, interpretación y transcripción de la información, lo que compromete la veracidad y confiabilidad de los datos obtenidos. En segundo lugar, la duplicidad de tareas implica una pérdida significativa de tiempo y recursos humanos, reduciendo la capacidad de la empresa en enfocar sus esfuerzos en la prevención de riesgos y en la mejora continua de sus procesos. En tercer lugar, la falta de herramientas digitales para el procesamiento de la información dificulta la obtención de reportes en tiempo real, restringiendo la posibilidad de identificar patrones de comportamiento y retrasando la implementación de medidas correctivas o preventivas. Finalmente, la ausencia de un sistema que vincule de manera directa las observaciones con planes de capacitación genera un desfase entre detección de conductas riesgosas y la respuesta formativa, debilitando la efectividad de los programas de seguridad laboral.

1.3 Breve descripción sobre la propuesta de solución.

Para abordar este desafío de ineficiencia y las limitaciones del sistema manual, se propone el desarrollo de un sistema web y móvil.

De esta necesidad nace CoreSafe, una plataforma diseñada para digitalizar y automatizar la gestión de los Programas de registro Conductual y las observaciones en faenas, permitiendo a los observadores y trabajadores de Indemin minimizar riesgos, automatizar procesos y optimizar la administración de la seguridad operacional.

En contexto de este trabajo se centrará únicamente en el desarrollo de la Plataforma Web de administración.

El proyecto incluye la creación de las siguientes partes:

- **Módulo de Gestión de PRC:** Permite crear, editar y organizar los distintos Programas de Registro Conductual. Incluye la posibilidad de generar nuevos PRC a partir de plantillas base personalizables, que pueden adaptarse a las necesidades de cada faena o tipo de observación.
- **Centralización de observaciones:** Reúne todas las observaciones registradas desde la aplicación móvil, permitiendo listar, filtrar y revisar cada registro en detalle.
- **Módulo de creación de plantillas:** Facilitará la creación y administración de plantillas que servirán como base para futuros PRC. Cada plantilla incluye nombre, descripción, secciones y preguntas configurables.
- **Módulo de administración del Sistema:** Este será un módulo administrativo para gestionar a los usuarios del sistema, pero también servirá como módulo de seguridad, ya que en este apartado se gestionarán los

permisos y roles que servirá como barrera que restringe el acceso a la información.

- **Módulo de notificaciones:** Informará automáticamente a la línea de mando cuando se haya realizado una observación.

1.4 Objetivos Generales y Específicos de la Tesina.

Desarrollar una plataforma web de administración modular y escalable, que permita digitalizar y optimizar la gestión de los Programas de Registro Conductual (PRC) y las observaciones en faenas, garantizando eficiencia, adaptabilidad y una experiencia de usuario fluida.

Los objetivos específicos que la aplicación web presenta son:

- Implementar una arquitectura modular y escalable para la plataforma web, utilizando Vue.js [1], que permita su adaptación a futuras necesidades y ampliaciones del sistema.
- Desarrollar la lógica de consumo de datos desde el *front-end* para interactuar eficientemente con la API del *back-end*, asegurando la correcta recuperación y sincronización de la información relacionada con los PRC y sus observaciones.
- Desarrollar la interfaz de usuario del módulo de plantillas, permitiendo crear, configurar y administrar de manera flexible los formatos base que servirán como referencia para los Programas de Registro Conductual (PRC)
- Desarrollar el módulo principal de gestión de PRC, permitiendo crear, visualizar y administrar los programas desde la plataforma web.
- Implementar las vistas de consultas necesarias para recuperar (desde el *back-end* central) y presentar en detalle las observaciones registradas por los observadores en la aplicación móvil.

1.5 Justificación de proyecto.

Desde el punto de vista técnico, este proyecto es esencial debido a que aborda un problema crítico en la gestión y administración de los programas de registros conductuales. Al implementar una plataforma web para la gestión de PRC, los usuarios finales se beneficiarán de una solución que les permitirá gestionar los PRC de manera más eficiente. La centralización y digitalización de la información de forma automatizada garantiza una respuesta más rápida y efectiva ante cualquier acción que se deba realizar.

El uso de tecnologías modernas, como Vue.js, permitirá una construcción eficiente y escalable del desarrollo *front-end* del sistema. Esto es fundamental para facilitar el acceso y la visualización de datos en un entorno que requiere actualizaciones constantes. Al integrar un sistema que digitaliza y automatiza acciones, logra minimizar los errores humanos y optimizar tiempos significativos en la gestión de la seguridad.

Beneficios:

Los beneficios esperados con este desarrollo son:

- **Eficiencia Operativa:** La automatización de procesos de registro de las observaciones eliminará errores manuales, optimizando el tiempo de gestión.

- **Usabilidad:** La interfaz fue diseñada para ser intuitiva, permitiendo a los trabajadores con diferentes niveles técnicos navegar sin dificultad, lo que facilitará la adopción del sistema, adaptando de formato físico a digital.
- **Experiencia de usuario mejorada:** Al ofrecer una digitalización del proceso, disminuye el tiempo de los trabajadores y automatizará el proceso.
- **Seguridad y Control:** Mediante el módulo de gestión de usuarios, el sistema permitirá gestionar el acceso, restringiendo información a los usuarios que no tengan los permisos adecuados.

Innovación:

Este proyecto incorpora diversas innovaciones tanto tecnológicas como metodológicas, que buscan mejorar la eficiencia, escalabilidad y experiencia de usuario de la plataforma:

- **Uso de librería moderna (Vue.js):** La adopción de Vue.js para el desarrollo *front-end* no solo garantiza una experiencia de usuario ágil y fluida, sino que también introduce una metodología de desarrollo basada en componentes, alineada con las mejores prácticas de ingeniería de software actual. Esto permite que el código sea más robusto, mantenible y fácil de testear.
- **Diseño progresivo y accesible:** La interfaz se diseñará con principios de accesibilidad, asegurando que sea intuitiva y usable para todos los administradores, independientemente de su nivel de experiencia tecnológica.
- **Optimización del rendimiento:** Se implementarán técnicas avanzadas (*lazy loading* y patrones de diseño) para minimizar los tiempos de carga y garantizar una navegación fluida, facilitando el trabajo diario de los usuarios y promoviendo un uso sostenido de la plataforma.
- **Arquitectura modular y escalable:** La plataforma se diseñará de forma modular, lo que permite agregar nuevas funcionalidades o módulos en el futuro sin afectar el funcionamiento general del sistema.

1.6 Metodología.

Para la construcción del *front-end* del sistema de CoreSafe se basó en la metodología ágil Scrum [2], seleccionada por su idoneidad en proyectos donde la adaptación a los requerimientos evolutivos y la retroalimentación continua son primordiales. Este marco de trabajo facilita la organización del esfuerzo en *sprints* de duración fija (típicamente de dos a tres semanas), para el contexto de este proyecto se organizó en 8 *sprints* de 2 semanas, desarrollados entre el 19 de julio y el 8 de noviembre de 2025, con reuniones de revisión realizadas con Indemin el miércoles posterior al cierre de cada sprint. La planificación general se estructuró en etapas breves asociadas a los ciclos de trabajo (enfocado en el desarrollo web): **(Inicio de proyecto)** Planificación inicial y arquitectura, **(1)** Configuración inicial del entorno de trabajo, **(2)** Definición y organización de la arquitectura base del proyecto web, **(3)** Desarrollo de funcionalidades de Login y Plantillas, **(4)** Desarrollo de funcionalidad de PRC, **(5)** Completar desarrollo de funcionalidad PRC, **(6)** Desarrollo de funcionalidad de Observaciones **(7)** Implementar validaciones y mejoras de calidad y **(8)** Funcionalidad de Usuarios y finalización del proyecto.

Al inicio de cada ciclo, se llevaron a cabo la planificación detallada para definir las tareas más prioritarias, las cuales se desglosaron en incrementos manejables enfocados en el diseño, desarrollo de la interfaz de usuario, y su posterior integración con los servicios del *back-end*. La colaboración constante con el equipo que se aseguró mediante reuniones diarias (en las primeras semanas de desarrollo) que posteriormente cambió a

reuniones semanales cuando los requerimientos ya estaban claros, mientras que la revisión del *sprint* al final de cada ciclo permitirá presentar los avances, recopilar *feedback* de los *stakeholders* y garantizar que el desarrollo permanezca alineado con los objetivos estratégicos de Indemin.

Para asegurar la calidad y la funcionalidad del *front-end*, se establecerá un proceso de Aseguramiento de la Calidad (QA) que involucra diversas etapas de *testing*. Inicialmente, se emplearán herramientas como Vue Test Utils [3] junto con su ejecutor como Vitest [4] para realizar pruebas unitarias de los componentes individuales, verificando su lógica y comportamiento aislado. Estas se complementarán con pruebas de integración y de extremo a extremo (E2E) mediante el *framework* Playwright [5], destinadas a validar la correcta interacción entre módulos y la respuesta de los servicios (incluida la conexión JWT [6]). Por último, se realizará una visita a terreno para evaluar el desempeño en un contexto de uso real, verificando su funcionamiento bajo las condiciones propias del entorno y asegurando que la experiencia del usuario final sea óptima.

La entrega incremental de funcionalidades permite iterar rápidamente y adaptarse a las necesidades cambiantes del proyecto, asegurando un *front-end* funcional, seguro y eficiente que cumpla con los objetivos de CoreSafe.

1.7 Breve descripción de la organización del informe en capítulos.

El presente documento está estructurado en capítulos que guían al lector a través del desarrollo del proyecto CoreSafe, desde su concepción hasta los resultados y conclusiones. A continuación, se describe brevemente el contenido de cada capítulo:

Capítulo 1: Introducción

Este capítulo contextualiza el proyecto, exponiendo sus antecedentes y la problemática que busca resolver, junto con una descripción general de la solución propuesta. Además, se definen los objetivos generales y específicos, se justifica la relevancia del trabajo y se describe la metodología empleada.

Capítulo 2: Marco Teórico

En este apartado se abordan los fundamentos teóricos del desarrollo *front-end*, analizando las tecnologías, librerías y herramientas seleccionadas. También se explica la arquitectura de software implementada, los patrones de diseño aplicados y su relevancia en la construcción del sistema.

Capítulo 3: Diseño e Implementación

Este capítulo describe el proceso de diseño e implementación del sistema, detallando la arquitectura, la organización de los componentes y las prácticas de desarrollo adoptadas. Asimismo, se incluyen aspectos relacionados con la codificación, los métodos de verificación, la resolución de errores y las estrategias de optimización utilizadas.

Capítulo 4: Conclusiones

Finalmente, se presentan las conclusiones del proyecto, destacando los aprendizajes, los resultados alcanzados y los principales desafíos enfrentados. Además, se propone recomendaciones y posibles mejoras para futuras versiones del sistema.

2 Marco teórico

2.1 Fundamentos del desarrollo front-end.

El *front-end* corresponde a la capa de software encargada de la interfaz con el usuario, es decir, la parte visible y directamente accesible dentro de una aplicación o sistema. Su propósito principal es permitir la interacción entre el usuario y la lógica del sistema, presentando de forma visual los datos procesados por el *back-end* y capturando las acciones del usuario para generar respuestas o solicitudes dentro de la aplicación. De acuerdo con Freeman (2020) [7], el *front-end* constituye el punto de contacto entre la experiencia humana y la funcionalidad técnica, siendo un elemento determinante en la usabilidad y eficiencia de los sistemas digitales.

El desarrollo del *front-end* tiene sus orígenes en los primeros años de la web, durante la década de 1990, cuando los sitios estaban compuestos principalmente por documentos HTML estáticos. Con la introducción de CSS (Cascading Style Sheets) en 1996, se mejoró la presentación visual y el control de los estilos. Posteriormente, JavaScript (1995) transformó la interacción en la web al permitir la manipulación dinámica del contenido del documento (Document Object Model, DOM), marcando el inicio de las interfaces interactivas (Flanagan, JavaScript: The Definitive Guide, 2020) [8].

A partir de la década de 2010, el surgimiento de *frameworks* y librerías (como Angular, React y Vue.js) revolucionó el desarrollo *front-end*, permitiendo la creación de páginas webs dinámicas de una sola página (*Single Page Applications*, SPA) con mayor modularidad, rendimiento y mantenibilidad. Estas herramientas promovieron un paradigma basado en componentes reutilizables, impulsando la productividad y consistencia en el desarrollo de interfaces complejas.

En la actualidad, el *front-end* continúa evolucionando hacia entornos multiplataforma y de alto rendimiento. Tecnologías como React Native, Ionic y Flutter permiten desarrollo de aplicaciones móviles nativas a partir de un único código base, reduciendo costos y tiempos de desarrollo. Asimismo, las *Progressive Web Apps* (PWAs) combinan las ventajas de las aplicaciones web y móviles, ofreciendo acceso sin conexión, notificaciones y mejor desempeño.

Las tendencias más recientes apuntan hacia la optimización del rendimiento, el diseño adaptable y la accesibilidad universal, utilizando técnicas como el *lazy loading* (carga perezosa), la renderización del lado del servidor (Server-Side Rendering, SSR) y el diseño *mobile-first*. Estas prácticas buscan mejorar la velocidad de carga, la adaptación a múltiples dispositivos y la inclusión de usuarios con diferentes capacidades.

El *front-end* se implementa en diversos tipos de sistemas, incluyendo:

- **Aplicaciones web:** Ejecutadas en navegadores mediante tecnologías estándar (HTML, CSS, JavaScript).
- **Aplicaciones móviles:** Desarrolladas con *frameworks* híbridos o nativos (React Native, Flutter, Ionic).
- **Aplicaciones de escritorio:** Mediante *frameworks* como Electron o Tauri

Esta versatilidad ha permitido expandir el ámbito del *front-end* más allá de la web tradicional destacando características como la interactividad, el diseño adaptable, la accesibilidad y la eficiencia en el rendimiento.

Entre sus ventajas destacan:

- Mejora de la experiencia del usuario mediante interfaces intuitivas.
- Adaptación a diferentes dispositivos y resoluciones.
- Posibilidad de actualizaciones rápidas sin afectar la lógica del servidor.
- Uso de herramientas y bibliotecas que agilizan el desarrollo.

Sin embargo, presenta desafíos importantes como:

- **Optimizar el rendimiento en entornos con recursos limitados:** Las interfaces modernas, aunque más atractivas e interactivas, exigen mayor capacidad de procesamiento, lo que puede afectar la velocidad y fluidez en dispositivos de menor rendimiento.
- **Compatibilidad entre navegadores:** Cada navegador utiliza motores de renderizado distintos, por lo que ciertas funciones pueden comportarse de manera diferente o no ser compatibles en versiones desactualizadas.
- **Pruebas y mantenimiento continuo:** La creciente interactividad de las aplicaciones incrementa la complejidad del *testing* y requiere actualizaciones constantes para asegurar un funcionamiento estable y compatible con nuevas tecnologías

2.2 Arquitectura de software.

El sistema se desarrolla bajo una arquitectura modular que se basa en la división de la aplicación en módulos independientes, cada uno de los cuales encapsula una funcionalidad específica. Este enfoque permite una mayor escalabilidad, mantenibilidad y reutilización de componentes, ya que cada módulo puede desarrollarse y probarse de manera aislada sin afectar al resto del sistema.

También favorece la adherencia de los principios SOLID [9], promoviendo que cada módulo cumpla con una única responsabilidad, sea abierto a extensión, pero cerrado a modificación, y dependa de abstracciones en lugar de implementaciones concretas, garantizando así la calidad de diseño y la sostenibilidad del sistema a largo plazo.

2.3 Patrón de diseño.

En el desarrollo del sistema se aplicaron diversos patrones de diseño que permiten estructurar el software de forma clara, modular y sostenible. Estos patrones contribuyen a separar responsabilidades, mejorar la mantenibilidad del código y facilitar la evolución futura del proyecto.

- **Data Store Pattern (Patrón de almacenamiento de datos):** Este patrón centraliza el estado de la aplicación en un único punto controlado, evitando la duplicación de información y garantizando coherencia entre los distintos módulos. Su principal ventaja es que simplifica el acceso y la actualización de datos compartidos, lo que facilita el mantenimiento y reduce errores asociados a estados diversos.

- **Service Layer Pattern (Patrón de capa de servicio):** Consiste en crear una capa dedicada para gestionar la comunicación con fuentes externas, como APIs. Se utiliza porque permite mantener la lógica de negocio separada de la presentación, mejorando la escalabilidad y la capacidad de prueba del proyecto
- **Humble Components Pattern (Patrón de componentes humildes):** Propone mantener los componentes o elementos de interfaz lo más simple posible, limitándolos a tareas de visualización.
- **Composable Pattern (Patrón composable):** Este patrón es la base de Vue 3, se basa en extraer la lógica reutilizable a unidades independientes que pueden compartirse entre distintos módulos o componentes. Se utiliza para mantener la cohesión del sistema y permitir que la lógica compleja evolucione sin afectar directamente la interfaz

2.4 Tecnologías y frameworks utilizados.

El desarrollo del *front-end* de la aplicación se realiza utilizando Vue 3 con Composition API [10], lo que permite estructurar la lógica de los componentes mediante funciones y referencias reactivas, favoreciendo la modularidad y reutilización del código. El proyecto se implementa en TypeScript [11], proporcionando tipado estático y mayor robustez, mientras que la gestión del estado global se realiza mediante Pinia [12], permitiendo un control centralizado y reactivo de los datos compartidos.

La interfaz combina Tailwind CSS [13], que facilita la construcción de diseños modernos mediante utilidades, y SCSS [14], que ofrece flexibilidad para la creación de estilos personalizados y el uso de variables, asegurando un código limpio y mantenible. Para la construcción y empaquetado se emplea Vite, garantizando un desarrollo ágil con recarga en caliente y optimización de recursos para producción. La comunicación con el *back-end* se realiza mediante la Fetch Api [15], optimizando las solicitudes y respuestas de manera eficiente.

El aseguramiento de la calidad del software se garantiza mediante pruebas unitarias con Vitest y Vue Testing Library, y pruebas end-to-end con Playwright, verificando el correcto funcionamiento de los flujos de la aplicación. En términos de la seguridad se refuerza mediante la protección de rutas basadas en roles y permisos definidos por el *back-end*.

Finalmente, el despliegue del *front-end* se realizará en los servidores propios de Indemin, en donde Felipe Gutiérrez (encargado de la Arquitectura de Software) será el responsable de levantar los servicios (*back-end* central y *front-end*) en los servidores.

3 Diseño del sistema

La definición de la arquitectura del sistema y de sus componentes fue el resultado de un análisis de los requerimientos funcionales, obtenidos mediante entrevistas y sesiones de retroalimentación con los trabajadores de Indemin. A partir de este proceso de levantamiento, fue posible identificar los principales casos de uso del sistema, los cuales permitieron determinar las funcionalidades esenciales y organizar su implementación en módulos claramente definidos.

3.1 Casos de uso

Los casos de uso identificados se agrupan en cinco procesos principales de interacción: autenticación del usuario, gestión de usuarios, gestión de programas de registro conductual, gestión de plantillas y gestión de observaciones.

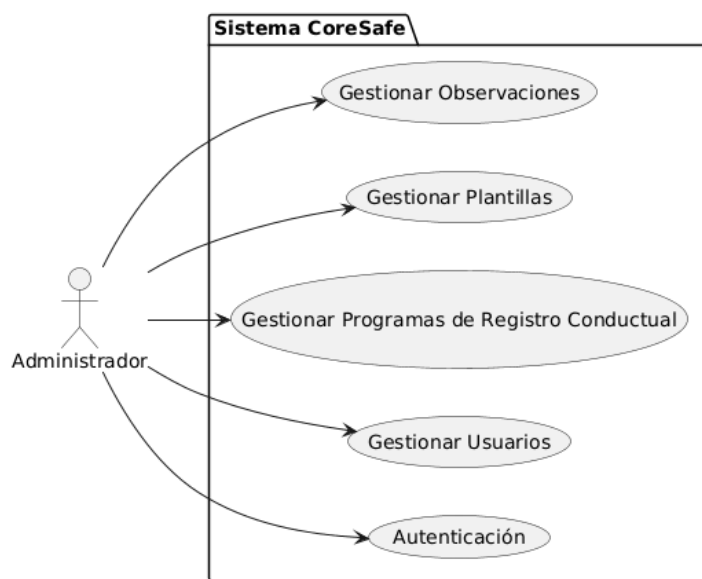


Ilustración 1: Diagrama de caso de uso general
Fuente: Elaboración propia

3.1.1 Autenticación

La gestión de autenticación permite que el administrador acceda de manera segura al sistema mediante las credenciales asociadas a su perfil. El sistema admite dos métodos de inicio de sesión: a través de una cuenta corporativa de Indemin o mediante un usuario administrador propio de la plataforma. Además, el caso de uso contempla la acción de cerrar sesión.

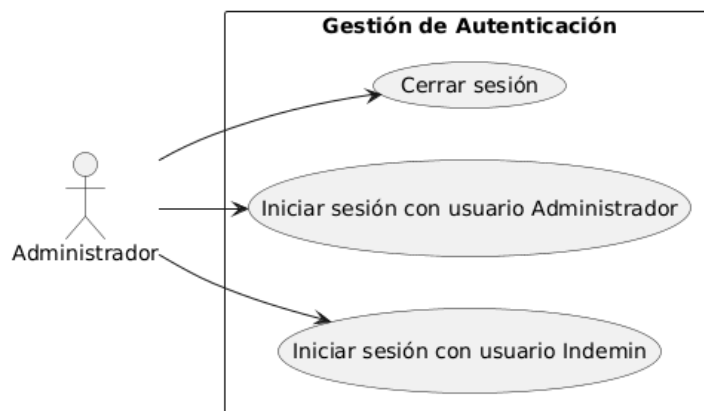


Ilustración 2: Diagrama de caso de uso: Gestión de Autenticación

Fuente: Elaboración propia

3.1.2 Gestión de Programas de Registro Conductual

La gestión de Programas de Registro Conductual (PRC) permite administrar todos los PRC. Este caso de uso incluye la visualización de los PRC existentes, la creación de nuevos registros a partir de plantillas, la revisión detallada de cada PRC, así como la modificación de su estado.

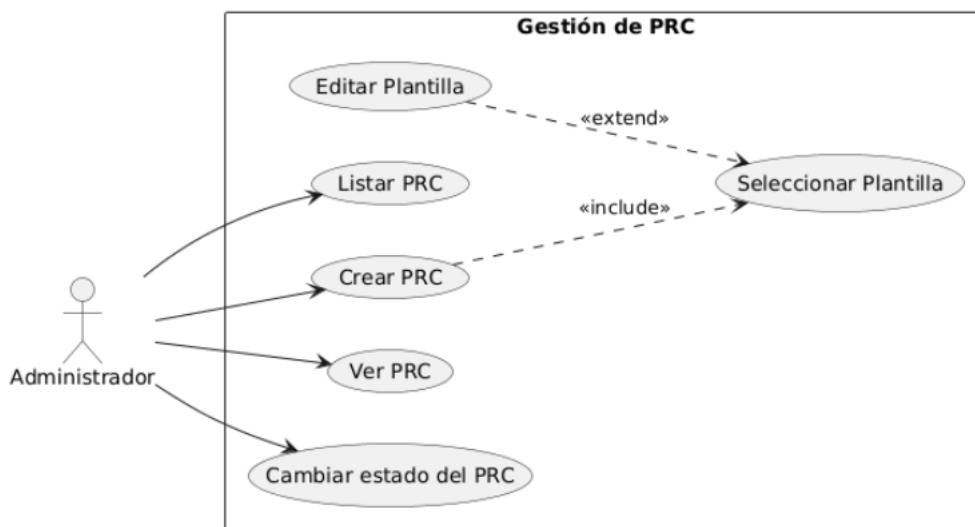


Ilustración 3: Diagrama de caso de uso: Gestión de PRC

Fuente: Elaboración propia

3.1.3 Gestión de Plantillas

La gestión de plantillas permite al administrador crear, visualizar, editar y eliminar plantillas que sirven como base para la construcción de Programas de Registro Conductual (PRC). Este proceso incluye la definición de la estructura de cada plantilla mediante la incorporación de secciones y preguntas configurables, lo que permite adaptarlas a las necesidades de cada faena o proceso de observación. Las plantillas creadas pueden ser reutilizadas y modificadas.

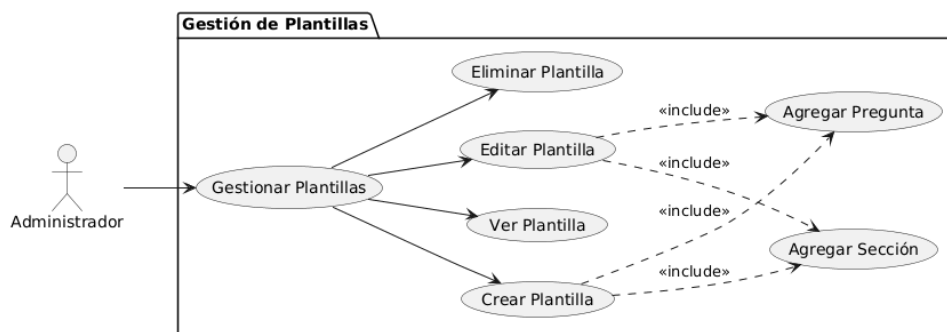


Ilustración 4: Diagrama de caso de uso: Gestión de Plantillas

Fuente: Elaboración propia

3.1.4 Gestión de Observaciones

La gestión de observaciones permite al administrador acceder a toda la información registrada en terreno asociada a un Programa de Registro Conductual (PRC). Este caso de uso incluye la visualización de todas las observaciones relacionadas a un PRC específico, así como la revisión detallada de cada registro, incluyendo las respuestas de sus formularios.

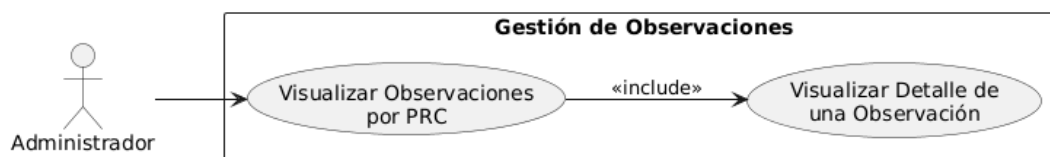


Ilustración 5: Diagrama de caso de uso: Gestión de Observaciones

Fuente: Elaboración propia

3.1.5 Gestión de Usuarios

La gestión de usuarios permite al administrador visualizar todas las cuentas registradas en la plataforma y administrar los roles asociados a cada usuario del sistema.

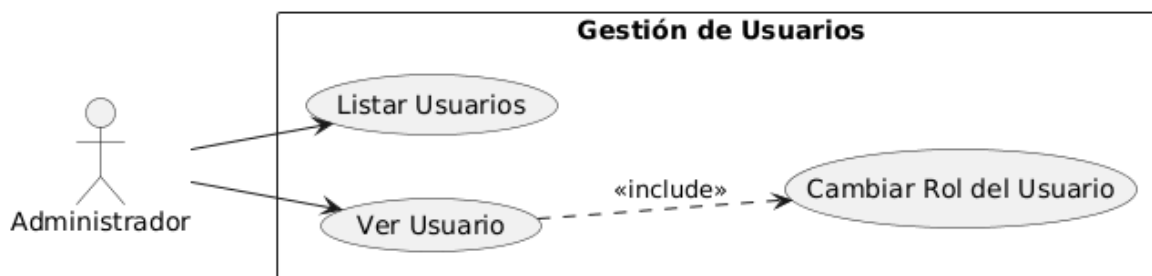


Ilustración 6: Diagrama de caso de uso: Gestión de Usuarios

Fuente: Elaboración propia

3.2 Arquitectura modular

En el desarrollo de aplicaciones web modernas, incluso en proyectos pequeños o de complejidad media (como el caso de CoreSafe), resulta fundamental adoptar una arquitectura definida que garantice la calidad y mantenibilidad del software. Aunque el *framework* Vue permite construir interfaces de manera rápida, la falta de una estructura clara puede generar desorden, duplicación de código y dificultades para escalar. Por ello, se utiliza una arquitectura modular [16] basada en funcionalidades, donde el sistema se organiza en unidades coherentes según dominios específicos.

Uno de los motivos principales para adoptar esta estructura es la posibilidad de distribuir responsabilidades de forma precisa, alineándose con los principios SOLID, particularmente con el principio de responsabilidad única (Single Responsibility Principle). Bajo este enfoque, cada módulo, componente, *store* o servicio cumple una función específica dentro del sistema, evitando que un mismo archivo concentre distintas tareas o lógicas que posteriormente dificulten su modificación. Este principio fomenta una mayor legibilidad, reduce el riesgo de introducir errores y permite que cada pieza del software pueda evolucionar de forma independiente.

Arquitectura Modular

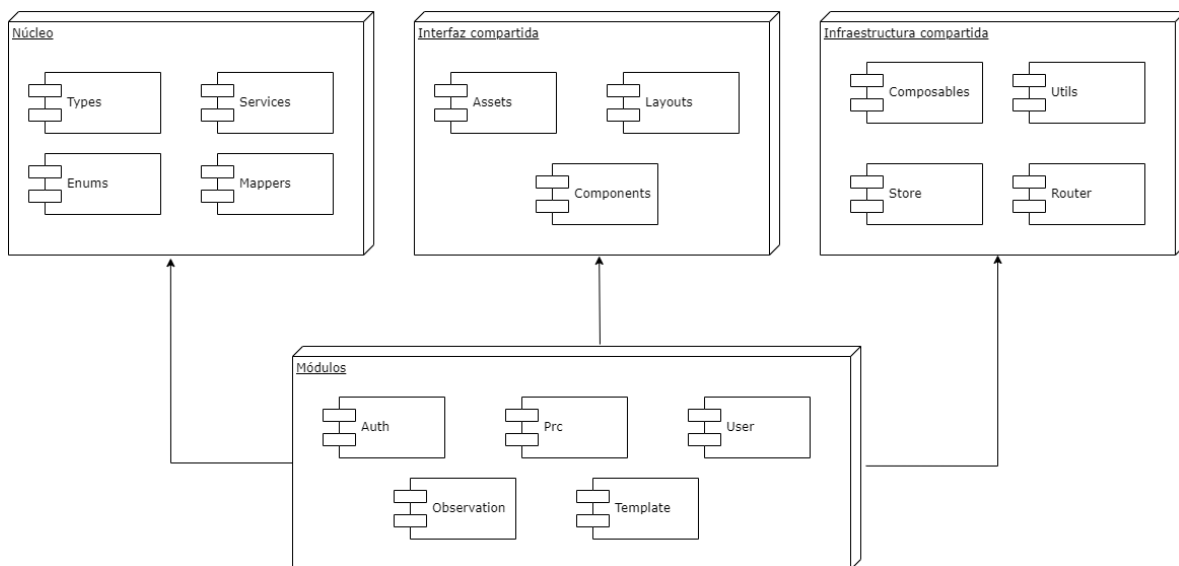


Ilustración 7: Arquitectura Modular
Fuente: Elaboración propia

3.2.1 Módulos

La carpeta "modules" constituye el núcleo de la arquitectura basada en funcionalidades. Cada módulo representa un dominio (lógica de negocio) específico del sistema y contiene de forma encapsulada los elementos necesarios para su funcionamiento, tales como componentes particulares, vistas y *composables*. Este enfoque permite mantener una alta cohesión interna en cada sección del proyecto y facilita la evolución independiente de cada funcionalidad, reduciendo el acoplamiento entre partes del sistema. Ejemplos de estos módulos son: "auth", "observation", "PRC", "template" y "users", cada uno organiza de acuerdo con sus propias necesidades operativas.

3.2.2 Recursos compartidos

A pesar de una separación modular, existen elementos que son comunes a toda la aplicación, como los componentes reutilizables, servicios de comunicación con la API, manejadores de estado compartido, utilidades o elementos gráficos. Estos recursos se ubican en carpetas compartidas, situadas fuera de los módulos, para garantizar que su uso sea accesible y homogéneo en todo el sistema.

3.2.2.1 Núcleo / Dominio

El Núcleo es la parte fundamental de la aplicación. Contiene la lógica transversal, los modelos, las reglas base y cualquier elemento que define cómo funciona el sistema a nivel general.

- Carpeta “*types*”: Se agrupan las interfaces, tipos y definiciones de datos utilizados en todo el sistema. Su función principal es garantizar consistencia en la tipificación, mejorar la legibilidad del código y prevenir errores derivados de estructuras de datos poco claras o implícitas. Esta capa contribuye directamente a la documentación interna del proyecto.
- Carpeta “*services*”: Representa la capa encargada de gestionar la comunicación con APIs externas (en este caso con la API principal).
- Carpeta “*enum*” permite definir un conjunto limitado y fijo de constantes con nombres que mejora la legibilidad y seguridad del código
- Carpeta “*mappers*” que contiene las funciones encargadas de transformar los datos entre diferentes capas del sistema (normalmente se usa para tener funciones que transforman datos del *back-end* al *front-end*)

3.2.2.2 Interfaz compartida

La interfaz compartida reúne todos los elementos visuales reutilizables. Su propósito es mantener una coherencia de diseño y evitar repetir componentes en cada módulo.

- Carpeta “*assets*”: Reúne recursos estáticos utilizados en distintas partes de la aplicación. Entre ellos se incluyen fuentes tipográficas, íconos, imágenes y estilos globales (css/scss).
- Carpeta “*components*” almacena componentes reutilizables no ligados a un dominio específicos. Estos elementos cumplen funciones transversales, como botones, formularios y tablas lo que permite integrarlos en múltiples módulos sin generar dependencias innecesarias y reduciendo la repetición de código.
- Carpeta “*layouts*” contiene las estructuras de páginas reutilizables que definen la distribución general de la interfaz, pero no el contenido específico de cada vista.

3.2.2.3 Infraestructura compartida

La infraestructura compartida reúne todos los elementos técnicos que permiten que la aplicación funcione, pero no son parte de la interfaz ni de la lógica de negocio.

Son herramientas y sistemas base que dan soporte operativo a toda la aplicación.

- Carpeta “*router*”: Se definen las rutas de la aplicación y la estructura de navegación, al interior de esta carpeta hay archivos por funcionalidad, separando responsabilidades.
- Carpeta “*stores*”: Concentra la gestión del estado global mediante Pinia, organizado en subcarpetas correspondientes a cada dominio del sistema. Esta separación permite mantener estados específicos de forma independiente.
- Carpeta “*utils*”: integra herramientas auxiliares que permite simplificar tareas transversales del proyecto.
- Carpeta “*composables*”: integra herramientas auxiliares que permite simplificar tareas transversales del proyecto, la diferencia con la carpeta “*utils*” es que la carpeta “*composable*” tiene lógica reactiva la cual es única en Vue.

3.2.3 Tests

La arquitectura modular también se refleja en la forma de organizar los tests. Cada módulo contiene sus propios casos de prueba, evitando concentrar la verificación del sistema en un único espacio. De esta manera, las pruebas se alinean con la estructura conceptual del proyecto: cada módulo valida su propio comportamiento y se responsabiliza de su correcto funcionamiento.

Además, se incluyen pruebas independientes para *composables* o funciones reutilizables, asegurando que los elementos transversales mantengan su integridad sin importar el contexto con el que se utilicen.

3.3 Estructura de los módulos

La estructura de los módulos constituye uno de los elementos centrales en la organización del sistema, dado que determina cómo se ordena la lógica interna, cómo se distribuyen las responsabilidades y cómo se facilita la escalabilidad del proyecto. Cada módulo funciona como un “bloque funcional coherente” dentro del sistema, es decir, representa un conjunto coherente de funcionalidades que responden a un mismo dominio del negocio.

A nivel conceptual, todos los módulos comparten una estructura interna similar, lo que favorece la homogeneidad y reduce la curva de aprendizaje para futuros desarrolladores. Esta estandarización garantiza que cada nuevo módulo integrado al sistema siga los mismos principios de diseño y facilite su mantenimiento.

Dentro de cada módulo, las unidades fundamentales son las vistas, los componentes y los *composables*.

Estructura de un módulo

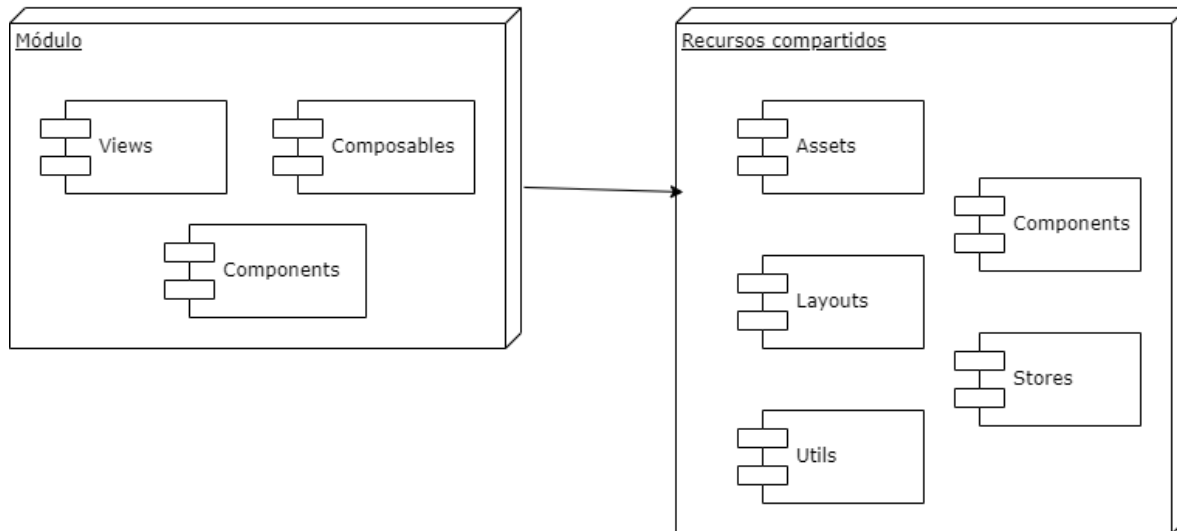


Ilustración 8: Estructura de un Módulo

Fuente: Elaboración propia

3.3.1 Views (Vistas)

Las vistas representan las páginas visibles para el usuario, siendo responsable de organizar la información y presentar los elementos gráficos necesarios para cada caso. Su papel es actuar como contenedores de la experiencia visual, dejando fuera cualquier lógica que no sea estrictamente relacionada con la presentación. Este enfoque permite que las vistas se mantengan simples y expresivas respecto a su propósito principal: mostrar información y recibir interacciones del usuario.

3.3.2 Components (Componentes)

Los componentes, por su parte, son piezas más pequeñas y reutilizables que operan dentro de las vistas. Su función es encargarse de elementos de interfaz que se repiten o que deben mantenerse aislados para facilitar su mantenimiento. Gracias a esta división, cada vista puede construirse combinando múltiples componentes especializados, manteniendo así una estructura clara y fácil de entender. Además, esta separación beneficia la consistencia visual del sistema, ya que elementos como tablas, botones o formularios mantienen el mismo comportamiento en todas las vistas.

3.3.3 Composables

Los *composables* representan la lógica de cada módulo. En ellos se concentran las funcionalidades que no son visuales, como obtener datos, manejarlos, transformarlos o ejecutar acciones que afectan al estado del sistema. Al trasladar la lógica a *composables* independientes, se logra que los módulos sigan el principio de responsabilidad única: cada parte del sistema se ocupa de una tarea claramente definida. Esta particularidad facilita la prueba individual de cada pieza, permite el reúso de funciones entre diferentes vistas e incluso simplifica la creación de nuevas funcionalidades sin duplicar código.

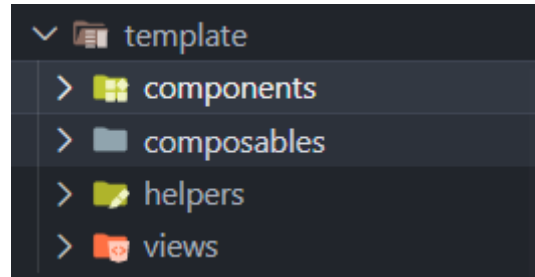


Ilustración 9: Estructura de un Módulo en el proyecto
Fuente: Elaboración propia

3.4 Descripción de los módulos

3.4.1 Módulo Auth (Autenticación y gestión de accesos)

El módulo *Auth* se encarga de gestionar el acceso al sistema y asegurar que solo usuarios autorizados puedan utilizar sus funcionalidades. Su diseño responde a la necesidad de integrar dos tipos de inicio de sesión: usuarios administradores propios de la plataforma y usuarios provenientes del ERP (sistema que integra y gestiona áreas de una empresa) de Indemin. Para resolver esta doble vía de autenticación, el módulo se estructura en torno a dos vistas principales: "*LoginAdministrador.vue*" y "*LoginIndemin.vue*", cada una orientada a su respectivo flujo de ingreso, se hizo de esta forma porque el usuario administrador requiere nombre y contraseña, en cambio el usuario Indemin requiere nombre y pin.

La lógica que sostiene ambos procesos se centraliza en el *composable* "*useAuth.ts*", encargado de gestionar la verificación de credenciales, controlar los estados de sesión y validar el rol del usuario. Dentro de esta validación, el sistema verifica que solo los usuarios con rol de administrador pueden ingresar. Si la persona inicia sesión desde el ERP, pero no cuenta con ese rol, el acceso es bloqueado igualmente.

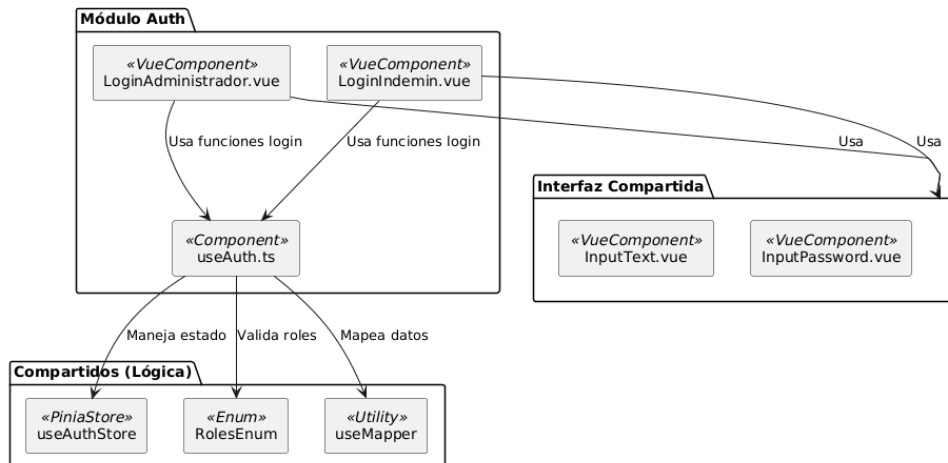


Ilustración 10: Diagrama de componentes "Módulo Auth"
Fuente: Elaboración propia

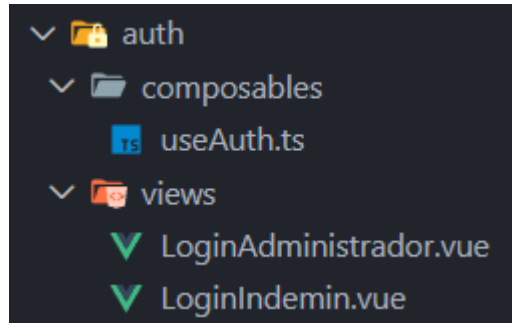


Ilustración 11: Estructura del “Módulo Auth” en el proyecto

Fuente: Elaboración propia

3.4.2 Módulo Observations (Observaciones)

El módulo de Observaciones está diseñado para gestionar la información asociada a las observaciones registradas dentro del sistema permitiendo listar todas las observaciones registradas y visualizar en detalle cada una de ellas. Para cumplir con este propósito, el módulo cuenta con dos vistas principales: “*ObservationList.vue*” y “*ObservationView.vue*”.

La vista “*ObservationList.vue*” es encargada de mostrar el listado general de las observaciones asociadas a los PRC. Esta presentación se realiza en una tabla que a su vez permite gestionar la eliminación de una observación cuando sea necesario. Para lograrlo esta vista se apoya en dos *composables*: “*useObservations.ts*”, que contiene la lógica que obtiene todas las observaciones, y “*useObservationDelete.ts*”, que concentra la lógica relacionada con el proceso de eliminación. Junto con ellos, también se utiliza el componente compartido “*DeleteModal.vue*”, encargado de presentar la confirmación visual antes de eliminar un registro.

Por otro lado, la vista “*ObservationView.vue*” se dedica exclusivamente a mostrar el detalle de una observación específica junto con las respuestas del formulario. Para obtener los datos necesarios, recurre a dos *composables*: “*useObservation.ts*”, responsable de recuperar la observación que se desea visualizar, y “*useObservationForm.ts*”, que obtiene el formulario asociado a dicha observación. Esta vista también utiliza el componente compartido “*Details.vue*”, que encapsula la lógica visual necesaria para mostrar toda la información de forma estructurada.

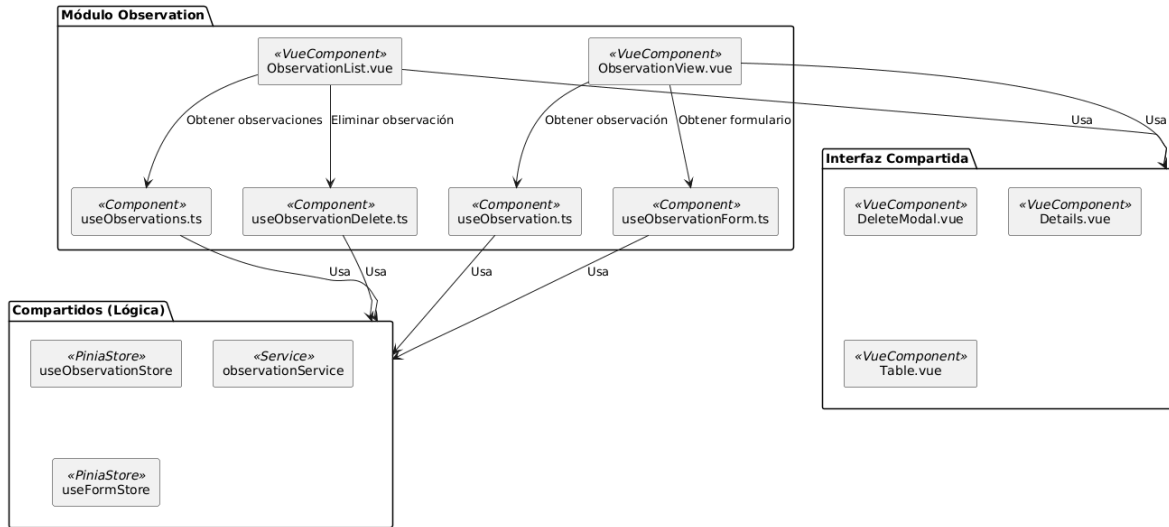


Ilustración 12: Diagrama de componentes "Módulo *Observations*"

Fuente: Elaboración propia

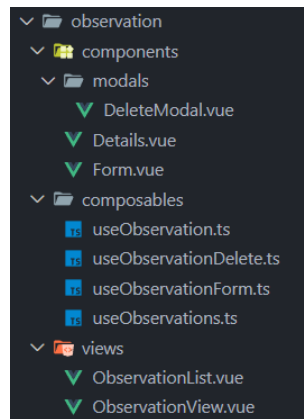


Ilustración 13: Estructura del "Módulo Observations" en el proyecto

Fuente: Elaboración propia

3.4.3 Módulo PRC (Programa de registros conductuales)

El módulo PRC concentra todo lo relacionado con la gestión de los Programas de registros conductuales (PRC). Su diseño busca mantener una estructura clara que permita, por un lado, visualizar el conjunto de registros existentes y, por otro, crear nuevos PRC o revisar en profundidad los ya registrados. Para lograrlo, el módulo reúne tres vistas principales "PRCList.vue", "PRCNew.vue" y "PRCView.vue".

La vista "PRCList.vue" cumple la función de presentar todos los PRC registrados en el sistema. El objetivo es ofrecer una visión global y ordenada, permitiendo al usuario desplazarse por el listado mediante paginación (utilizado como medida de optimización). Esta vista se apoya en el componente reutilizable "Table.vue", encargado de mostrar los datos de una tabla uniforme en toda la plataforma.

El comportamiento dinámico del listado (como obtener los registros desde el *back-end* o actualizar la página actual) es administrado por el *composable* "usePRCs.ts", que reúne

la lógica necesaria para manejar el flujo de datos sin sobrecargar la vista. De esta manera, la interfaz se mantiene liviana y centrada únicamente en presentar la información. Además, en esta vista se puede manejar el estado de un PRC, cambiando a ser "Finalizado" cuando la línea de mando lo estime conveniente.

Cuando el usuario selecciona un PRC específico, el sistema lo redirige a "PRCView.vue", la vista dedicada a mostrar un PRC en detalle. Aquí, el propósito es brindar una lectura completa del contenido, sin permitir edición. Esta vista utiliza el *composable* "usePRC.ts", que realiza la obtención del PRC según su identificador y prepara la información para ser mostrada. Para mantener una presentación consistente, la vista integra el componente "Details.vue", encargado de organizar los datos del PRC.

La vista "PRCNew.vue" está enfocada exclusivamente en el proceso de creación de un nuevo PRC. Su función es preparar todo lo que el usuario necesita para completar el formulario, validar la información ingresada y enviarla al sistema.

El formulario que se muestra en esta vista se construye a partir del componente reutilizable "Form.vue", el cual proporciona una estructura uniforme para la entrada de los datos en todo el proyecto. Para completar este formulario, la vista se apoya en varios *composables* con responsabilidades específicas:

- "usePRCForm.ts": Reúne los datos necesarios para llenar los selectores y campos dinámicos del formulario, además de coordinar el envío de la información hacia la API.
- "usePRCValidationForm.ts": Encapsula las reglas de validación, permitiendo verificar que los campos estén correctamente completados antes de enviar el PRC.
- "useTemplatePRC.ts": Administra la selección y eventual edición de una plantilla, facilitando que el usuario parta de un conjunto predefinido de elementos cuando corresponda.

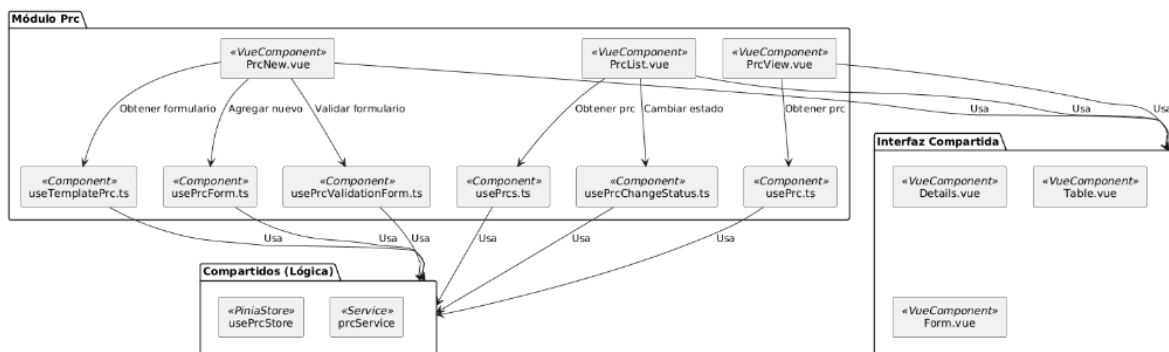


Ilustración 14: Diagrama de componentes "Módulo PRC"

Fuente: Elaboración propia

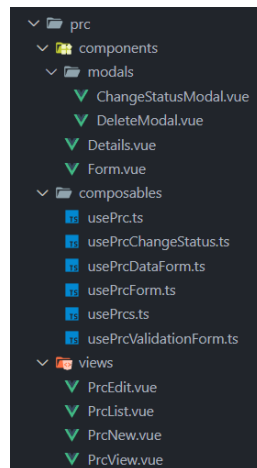


Ilustración 15: Estructura del “Módulo PRC” en el proyecto

Fuente: Elaboración propia

3.4.4 Módulo Template (Plantillas)

El módulo *Template* es uno de los más amplios del sistema, ya que se encarga de gestionar las plantillas que luego sirven como base para la creación de los PRC. A diferencia de los otros módulos, aquí se incorpora una carpeta adicional llamada “*helpers*”, que contiene funciones destinadas a facilitar la creación y organización de las estructuras internas de las plantillas.

Este módulo está compuesto por cinco vistas principales, cada una orientada a una fase diferente del ciclo de vida de una plantilla.

La primera vista es “*TemplateList.vue*”, encargada de mostrar todas las plantillas disponibles. Aquí se ofrece paginación, opciones para eliminar una plantilla, así como accesos directos para editarla o ver su detalle. Esta vista se apoya en el *composable* “*useTemplates.ts*”, que coordina la obtención de la información, y presenta los datos mediante el componente reutilizable “*Table.vue*”, lo que mantiene una uniformidad visual en todo el sistema.

La segunda vista es “*TemplateView.vue*”, destinada a mostrar en detalle completo de una plantilla. Para lograrlo combina los componentes “*Details.vue*” y “*Section.vue*”, que permite visualizar de manera clara tanto los datos generales como las secciones que componen cada plantilla. Esta vista utiliza el *composable* “*useTemplate.ts*”, responsable de obtener la información específica de la plantilla seleccionada.

La tercera vista es un “*TemplateNew.vue*”, enfocada en la creación de nuevas plantillas. Aquí reúnen todos los elementos necesarios para definir una estructura completa y flexible. La vista utiliza el *composable* “*useTemplates.ts*”, encargado de procesar la creación, y el *composable* “*useTemplateValidation.ts*”, que concentra las reglas de validación necesarias para asegurar que la plantilla creada corresponda con los campos requeridos por la API. Además, hace uso de los “*helpers*”: “*createNewTemplateForm.ts*” y “*createNewSection.ts*” para construir de manera dinámica tanto la estructura general como sus secciones. En cuanto a los componentes de apoyo, se emplean

"NewDetails.vue", "NewSection.vue" y el botón "AddSection.vue", los cuales permiten guiar al usuario en la construcción ordenada de cada parte de la plantilla.

La cuarta vista es "TemplateEdit.vue", que permite editar una plantilla existente. Para esto, obtiene los datos mediante el *composable* "useTemplate.ts", valida los cambios con "useTemplateValidation.ts", y utiliza nuevamente los componentes reutilizables "NewDetails.vue", "NewSection.vue" y "AddSection.vue", manteniendo una experiencia similar a la creación de una plantilla nueva, pero orientada a la modificación de una ya existente.

Finalmente, está "TemplatePRCEdit.vue", una vista creada a partir de una necesidad particular: permitir la edición de una plantilla asociada específicamente a un PRC, sin modificar la plantilla original. Esto permite tener plantillas únicas y adaptadas a cada caso, sin interferir con las plantillas base del sistema. Para ello, la vista obtiene la información mediante el *composable* "useTemplatePRC.ts", y utiliza "useTemplateValidation.ts" para asegurar que los cambios cumplan con las reglas definidas. Al igual que en las vistas anteriores, se apoya en los componentes "NewDetails.vue" y "NewSection.vue" para presentar y permitir modificar la información de manera clara y estructurada.

Cabe destacar que el componente "NewSection.vue" incluye lógica interna que se apoya en los *composables* "useFormError.ts" y "useFormErrorQuestion.ts", los cuales entregan mensajes de error cuando existe algún problema en la información ingresada.

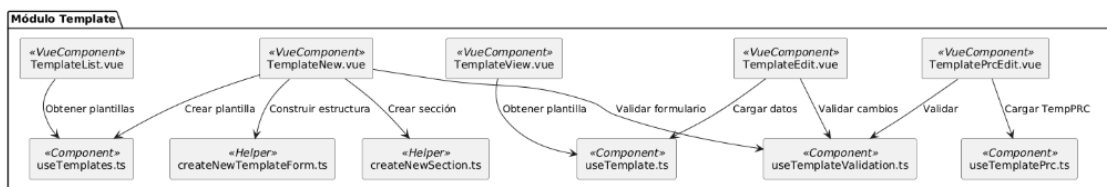


Ilustración 16: Diagrama de componentes "Módulo Template"

Fuente: Elaboración propia

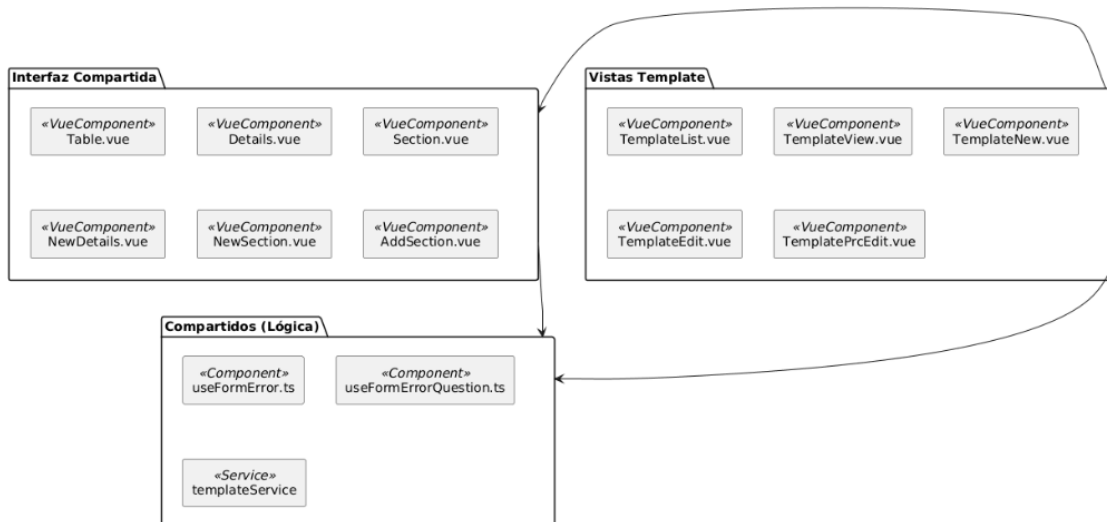


Ilustración 17: Diagrama de componentes "Módulo Template"

Fuente: Elaboración propia

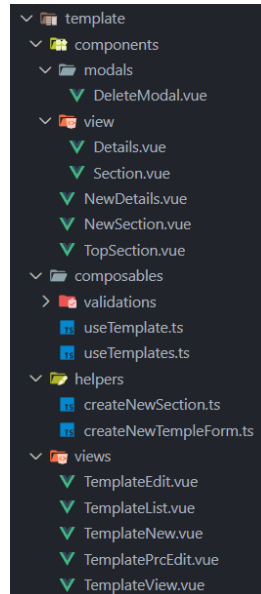


Ilustración 18: Estructura del “Módulo Template” en el proyecto

Fuente: Elaboración propia

3.4.5 Módulo Users (Usuarios)

El módulo de Usuarios cumple una doble función dentro del sistema: por un lado, permite visualizar el listado completo de las cuentas registradas, y por otro, ofrece la posibilidad de modificar el rol asignado a cada usuario. Ambas tareas se desarrollan dentro de una única vista, “*UsersList.vue*”, lo que responde a la intención de mantener este módulo lo más simple y directo posible, dado que su propósito es complementario a la gestión principal del sistema.

En esta vista se presenta una tabla paginada donde se muestran todos los usuarios disponibles. La vista se apoya en el *composable* “*useUsers.ts*”, encargado de obtener los datos que alimentan este listado, y en el *composable* “*useMyUser.ts*”, que permite identificar al usuario actualmente autenticado. Esta última información resulta relevante para restringir acciones (no poder editarse su propio rol).

En cuanto a la modificación de roles, esta funcionalidad se realiza mediante un componente separado, “*ChangeRoleModal.vue*”, pensado para no interrumpir el flujo del listado y para presentar la acción en un entorno controlado. El modal se activa desde la tabla. Para ejecutar esta acción, la vista utiliza el *composable* “*useChangeRole.ts*”, que encapsula la lógica necesaria para actualizar esta información.

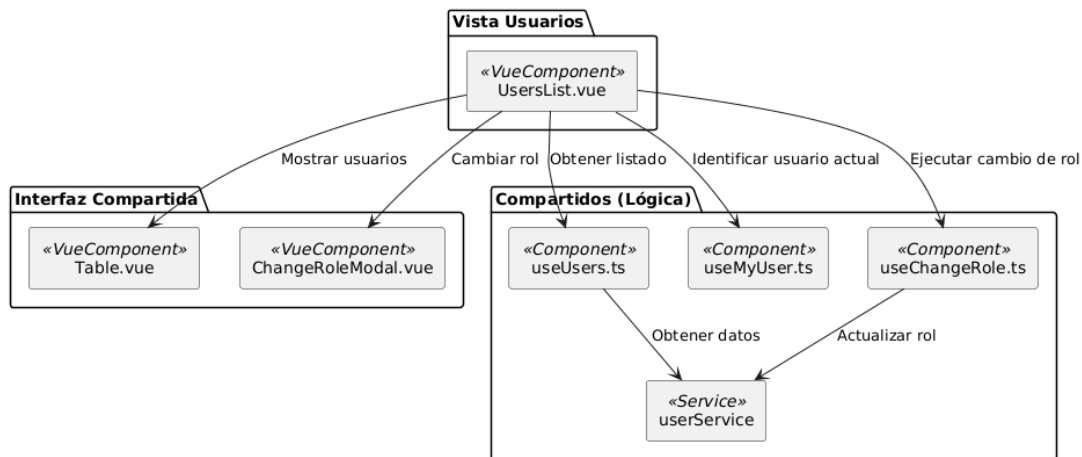


Ilustración 19: Diagrama de componentes “Módulo Users”
Fuente: Elaboración propia

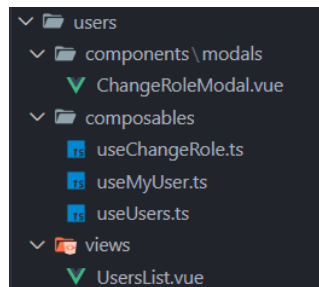


Ilustración 20: Estructura del “Módulo Users” en el proyecto
Fuente: Elaboración propia

3.5 Patrones de diseño y buenas prácticas

Los patrones de diseño constituyen soluciones probadas a problemas recurrentes en la construcción de software. Se basa en la experiencia acumulada de la ingeniería de software y ofrecen guías estructurales o conceptuales que hacen posible desarrollar sistemas más robustos, mantenibles y fáciles de evolucionar. En lugar de imponer una arquitectura fija, los patrones permiten adoptar decisiones que se ajustan al contexto del proyecto, facilitando la estandarización de buenas prácticas y evitando la reinención constante de las soluciones.

En el mismo proyecto es habitual la coexistencia de múltiples patrones de diseño, debido a que cada uno aborda una dimensión específica del problema: gestión del estado, composición de lógica, interacción con APIs externas o estructuración de componentes visuales. La aplicación combinada de patrones contribuye a la modularidad y claridad del código, permitiendo que la lógica de negocio, la presentación y la comunicación con servicios se mantenga debidamente separadas. Este enfoque incrementa la cohesión interna de cada módulo y reduce el acoplamiento entre sus partes, lo que se traduce en un software más estable y fácil de mantener en el tiempo.

3.5.1 Uso de Lazy Loading en Vue 3

En el diseño del sistema se implementó *lazy loading* exclusivamente a nivel de enrutamiento, utilizando las capacidades provistas por Vue Router (biblioteca oficial de Vue encargada de la gestión de rutas y navegación entre vistas). Esta estrategia permite que las vistas asociadas a cada ruta se carguen de forma diferida, es decir, únicamente cuando el usuario accede a ellas, contribuyendo a una reducción del tiempo de carga inicial de la aplicación y a un uso más eficiente de los recursos del navegador.

No se consideró necesario aplicar *lazy loading* a nivel de componentes individuales, dado que la arquitectura modular adoptada y el uso de patrones de diseño como *Humble Components*, *Composable Pattern* y *Service Layer Pattern* permitieron una separación clara de responsabilidades. Gracias a esta organización, los componentes de interfaz mantienen un tamaño reducido y una lógica acotada, evitando cargas innecesarias y haciendo innecesaria una optimización adicional mediante carga diferida de componentes.

3.5.2 Data Store Pattern (Patrón de almacenamiento de datos)

El “*Data Store Pattern*” es un patrón de diseño orientado a la gestión centralizada y coherente del estado de una aplicación. Su propósito principal es aislar la lógica relacionada con el almacenamiento, la actualización y la obtención de datos, evitando que dicha responsabilidad recaiga directamente en los componentes de la interfaz. Este patrón establece un punto único y consistente para mantener el estado, permitiendo que cada módulo del sistema interactúe con los datos mediante una capa dedicada, en lugar de acceder a ellos de forma dispersa o no controlada.

En el caso del presente proyecto, este patrón se implementa utilizando Pinia, el administrador de estado oficial de Vue 3, que proporciona un mecanismo declarativo y reactivo para crear *stores* modulares. Un ejemplo de esto es el “*auth.store.ts*”, encargado de conservar la información del usuario autenticado. Su diseño es simple: mantiene un estado (“*user*”) y expone métodos de mutación (“*setUser()*”, “*clearUser()*”), garantizando que toda la lógica relacionada a la identidad del usuario quede encapsulada en un único punto del sistema.

```
import { defineStore } from 'pinia';
import { ref } from 'vue';

import { type User } from '@core/types/auth/user';

export const useAuthStore = defineStore('auth-store', () => {
  const user = ref<User|null>(null);

  const setUser = (newUser: User) => user.value = newUser;
  const clearUser = () => user.value = null;

  return {
    user,
    setUser,
    clearUser
  }
});
```

Ilustración 21: Extracto de código de Store “Auth”
Fuente: Elaboración propia

Junto al store, se utiliza un *composable* “*useMyUser.ts*” que actúa como una capa de acceso al estado centralizado. Este *composable* recupera únicamente los datos que necesita el consumidor (en este caso, el usuario) sin exponer la totalidad del store ni su lógica interna. Con ello se refuerza el principio de responsabilidad única: el store administra el estado, mientras que el *composable* distribuye de manera controlado ese estado hacia la interfaz.

```
import { useAuthStore } from "@stores/auth.store"
import { storeToRefs } from "pinia";

const useMyUser = () => {

  const store = useAuthStore()
  const {user} = storeToRefs(store);

  return {
    user,
  }
}

export default useMyUser;
```

Ilustración 22: Extracto del *composable* “*useMyUser.ts*”
Fuente: Elaboración propia

Todo este enfoque cobra sentido al analizar un caso práctico: la existencia de una función que se ejecuta durante el proceso de autenticación del usuario. Esta función (invocada al hacer login) obtiene la información del usuario desde el servicio correspondiente, la transforma mediante un “*mapper*” y finalmente la almacena en el store mediante “*setUser()*”. Gracias a ello, la aplicación no necesita volver a realizar solicitudes para recuperar los datos del usuario en cada vista o componente. Cualquier módulo que requiera esta información solo debe acceder al store a través del *composable*, obteniendo de inmediato el estado persistido en memoria. Esto reduce llamadas redundantes, elimina duplicación de lógica y garantiza que toda la interfaz responda consistentemente al mismo estado centralizado.

```
export function useAuth() {

  const store = useAuthStore()
  const {user} = storeToRefs(store);

  const loading = ref(false);
  const errorLogin = ref('');
  const errorIndemin = ref(false);

  const me = async() => {
    try{
      const token = localStorage.getItem('token');

      const response = await fetchMe();

      if(!response) {
        await logout();
        throw new Error('Token invalido.');
      }

      const data = response;

      const userData: User = userMapper({...data, access_token: token})

      store.setUser(userData);

    }catch(err: any){
      console.log(err);
    }finally{
    }
  }
}
```

Ilustración 23: Extracto del *composable* “*useAuth.ts*”
Fuente: Elaboración propia

3.5.3 Service Layer Pattern (Patrón de capa de servicio)

El “*Service Layer Pattern*” se utiliza para organizar y centralizar la interacción con fuentes externas de datos (como APIs REST [18]) mediante una capa de servicios que actúa como intermediaria entre la lógica de la aplicación y los *endpoints* remotos. Su propósito es encapsular el acceso a datos, estandarizar la comunicación con el *back-end* y evitar que las vistas o *composables* dependan directamente de llamados HTTP (Protocolo que usan los navegadores y servidores para comunicarse con la web). Esta separación incrementa la mantenibilidad, facilita la escalabilidad y permite modificar la fuente de datos sin afectar las capas superiores.

En el proyecto, este enfoque se desarrolla a través de un conjunto de funciones alojadas en la carpeta *services*, cada una responsable de la operación específica del dominio. Un ejemplo ilustrativo es el archivo “*observation.service.ts*”, donde se define la función “*fetchGetObservationsByPrc()*”. Esta función concentra la llamada HTTP al *back-end*, administra el manejo básico de errores y devuelve los datos ya procesados para que el resto del sistema solo reciba información limpia y estandarizada.

```
export const fetchGetObservationsByPrc = async (prcId: number, page: number = 1): Promise<ResponsePagination<Observation>>=> {  
  const response = await fetch(`${API_URL}/observations/?prc_id=${prcId}&page=${page}`, {  
    method: "GET",  
    headers: headers(),  
  });  
  
  if (!response.ok) {  
    throw new Error("Error al obtener las observaciones");  
  }  
  return await response.json();  
};
```

Ilustración 24: Extracto del *service* “*observation.service*”
Fuente: Elaboración propia

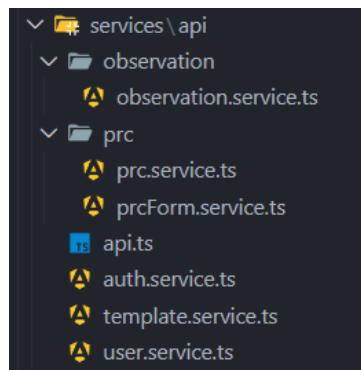


Ilustración 25: Estructura de la carpeta “*service*” en el proyecto
Fuente: Elaboración propia

La función se limita exclusivamente a su responsabilidad: obtener información desde la API. No administra estado, no muta datos internos del sistema ni realiza transformaciones complejas. De este modo, se cumple con el principio de responsabilidad única (SRP), evitando mezclar tareas de red con operaciones de negocio o manipulación de estado.

El valor del “*Service Layer Pattern*” se hace evidente cuando esta capa se integra con los *composables*, que funcionan como la lógica de orquestación de cada módulo. En el *composable* “*useObservations.ts*”, por ejemplo, la función “*getObservations()*” invoca a

"*fetchGetObservationsByPRC()*", recibe los datos ya validados y los utiliza para actualizar el estado global mediante el store.

```
const getObservations = async (id:number) => {
  loadingObservation.value = true
  try{
    store.setObservationList([]);
    const response = await fetchGetObservationsByPrc(id, pagination.value.current_page);
    store.setObservationList(response.data);
    pagination.value = mapResponsePaginationToMeta(response.meta);
  }catch(error){
    throw error;
  }finally{
    loadingObservation.value = false
  }
}
```

Ilustración 26: Extracto del *composable* "useObservations.ts"

Fuente: Elaboración propia

Para complementar este patrón, se definió un archivo llamado "*response.ts*", cuyo objetivo es estandarizar la estructura de datos que retorna cada servicio. Este archivo contiene las interfaces "*ResponsePagination*", "*Meta*" y "*Response*", que garantizan que todos los servicios del sistema devuelven un formato uniforme

3.5.4 Humble Components Pattern (Patrón de componentes humildes)

El "*Humble Components Pattern*" propone que los componentes de interfaz se mantengan deliberadamente "humildes", es decir, simples, predecibles y libres de lógica compleja. Su función principal es recibir datos ya procesados y mostrarlos en pantalla, delegando toda la lógica de validación, transformación o comunicación con el *back-end* hacia capas externas, como servicios, *composables*, *stores* o componentes controladores. Esto reduce el acoplamiento, favorece la reutilización y facilita las pruebas, ya que los componentes quedan centrados exclusivamente en la presentación.

```
<script setup lang="ts">
import { type Section } from '@core/types/template/template';

defineProps<{
  section: Section,
}>();
</script>

<template>
  <div class="card card-template">
    <div>
      <p class="font-bold">{{section.name}}</p>
    </div>
    <div class="flex gap-2" v-for="(question,index) in section.form_questions" :key="question.id">
      <p>{{ index + 1 }}</p>
      <p>{{ question.question }}</p>
    </div>
  </div>
</template>
```

Ilustración 27: Componente "*Section.vue*"

Fuente: Elaboración propia

En el proyecto, este patrón se observa claramente en el componente “*Section.vue*”. Este componente recibe un objeto “*section*” a través de sus *props* (*propiedades que un componente recibe desde otro componente*) y se limita a renderizar su contenido, mostrando el nombre de la sección y su lista de preguntas. No ejecuta llamadas a la API, no gestiona estados globales ni locales complejos, ni transforma datos. Su responsabilidad se restringe al mercado y a la visualización, lo que ejemplifica con precisión la idea de un componente humilde.

```
<template>
  <section class="template" v-if="template">
    <TopSection title="Plantilla" text="Editar" @click="router.push({name:'templates-edit',params:{id:templateId}})" />
    <Details :template="template" />
    <template v-for="(section) in template.form_sections" :key="section.id" >
      <Section :section="section" />
    </template>
  </section>
  <section class="flex justify-center items-center" v-else>
    <IconLoaderPage />
  </section>
</template>
```

Ilustración 28: Extracto de componente *TemplateView.vue*
Fuente: Elaboración propia

La lógica necesaria para obtener y preparar la información se ubica fuera del componente, específicamente en la vista “*TemplateView.vue*”, donde se invoca el *composable* “*useTemplate.ts*” para cargar la plantilla y sus secciones. De esta manera, “*TemplateView.vue*” actúa como coordinador: obtiene los datos, maneja el ciclo de vida y distribuye la información a los componentes de presentación. “*Section.vue*”, en cambio, solo recibe los datos ya listos para ser mostrados, manteniendo así una separación clara entre lógica y presentación.

3.5.5 Composable Pattern (Patrón composable)

El “*Composable Pattern*” constituye una de las prácticas más relevantes en el diseño de aplicaciones con Vue 3, ya que permite encapsular y reutilizar la lógica de negocio de manera completamente independiente de los componentes visuales. Su propósito central es evitar que los componentes asuman responsabilidades que no les corresponden (como la gestión del estado, las validaciones o la comunicación con servicios externos), promoviendo así el principio de responsabilidad única y una estructura más mantenible.

En el ejemplo del módulo *Template*, el componente “*TemplateNew.vue*” utiliza distintos *composables*: “*useTemplates.ts*” y “*useTemplateValidation.ts*”. Cada uno encapsula una parte específica de la lógica: creación de una nueva plantilla y validación de los campos. Gracias a esta separación, la vista mantiene un rol estrictamente declarativo, limitándose a orquestar acciones, escuchar eventos y renderizar el estado expuesto por estos *composables*. Toda la lógica compleja (como llamadas asíncronas, validaciones, reactividad interna y transformación de datos) permanece aislada en funciones especializadas y reutilizables.

```
const {createSuccess, addTemplate} = useTemplates();  
const {validateTemplate} = useTemplateValidation()
```

Ilustración 29: Extracto de componente “*TemplateNew.vue*”

Fuente: Elaboración propia

3.6 Integración con el sistema general

La integración entre el sistema desarrollado y el resto de la plataforma se realiza mediante comunicación HTTP, utilizando solicitudes y respuestas en formato JSON (formato de texto para guardar o enviar datos), lo que permite mantener un intercambio de datos simple y estandarizado.

Para organizar esta interacción, se aplicó el “*Service Layer Pattern*”, que consiste en concentrar la comunicación con la API en una capa independiente y claramente definida. Gracias a esta separación, la lógica de la interfaz no accede directamente al *back-end*, sino que lo hace exclusivamente a través de funciones de servicio. Esto simplifica el mantenimiento, evita duplicación y garantiza que los cambios futuros afecten solo a un punto centralizado.

3.6.1 Funcionamiento de una función de servicio

Cada función del servicio sigue una estructura uniforme:

- Ejecución de la petición con *fetch* (función que sirve para hacer solicitudes HTTP), enviando o recibiendo datos en formato JSON.
- Construcción de la URL utilizando la constante `API_URL`, que centraliza la dirección del *back-end*.
- Definición del método HTTP correspondiente (GET, POST, PUT o DELETE).
- Uso del objeto obtenido en la función “*headers()*” (que se explicará más adelante), que incorpora el token de autenticación junto con la configuración estándar requerida.
- Validación de la respuesta, verificando si la solicitud fue exitosa.
- Retorno de los datos, utilizando estructuras con tipos definidos que permiten trabajar con respuestas consistentes en el resto de la aplicación.

Este patrón se repite en todos los servicios, manteniendo una lógica clara, predecible y sencilla de mantener. Un ejemplo representativo es el siguiente:

```
import { API_URL, headers } from "../api";...

export const fetchGetObservation = async (observationId: number): Promise<Response<Observation>> => {
  const response = await fetch(`${API_URL}/observations/${observationId}`, {
    method: "GET",
    headers: headers(),
  });

  if (!response.ok) {
    throw new Error("Error al obtener la observación");
  }
  return await response.json();
};

export const fetchGetObservationsByPrc = async (prcId: number, page:number=1): Promise<ResponsePagination<Observation>>=> {
  const response = await fetch(`${API_URL}/observations/?prc_id=${prcId}&page=${page}`, {
    method: "GET",
    headers: headers(),
  });

  if (!response.ok) {
    throw new Error("Error al obtener el formulario");
  }
  return await response.json();
};
```

Ilustración 30: Extracto del service “*observation.service*”**Fuente:** Elaboración propia

3.6.2 Función “*headers()*”

Todas las solicitudes HTTP utilizan una función común encargada de preparar los encabezados necesarios. Esta función agrega:

- El token almacenado en el navegador, requerido para la autorización.
- El tipo de contenido, estandarizado como JSON

Gracias a esta función, se asegura que todas las llamadas a la API se realicen bajo un esquema uniforme y seguro.

```
src > services > api > api.ts > ...
1   export const API_URL = import.meta.env.VITE_API_URL
2
3
4   export const headers = () => {
5     const token = localStorage.getItem("token");
6     return {
7       "Content-Type": "application/json",
8       Authorization: `Bearer ${token}`,
9     }
10  }
```

Ilustración 31: Función “*headers*”**Fuente:** Elaboración propia

3.6.3 Tipos de retorno

Para mantener consistencia en todas las comunicaciones, el sistema utiliza dos tipos principales de respuesta: “*Response*” y “*ResponsePagination*”. Ambos están definidos en el archivo “*response.ts*” y están contruidos sobre un genérico, lo que permite que los datos retornados puedan adaptarse a cualquier modelo sin repetir estructuras.

Se utiliza *Response*<T> cuando la API devuelve un único elemento o un conjunto no paginado y se utiliza *ResponsePagination*<T> cuando el *back-end* envía información paginada. Incluye un arreglo y un objeto con metadatos de paginación.

```
src > core > types > response.ts > ...  
1  export interface ResponsePagination<T> {  
2    data: T[],  
3    meta: Meta  
4  }  
5  
6  export interface Meta {  
7    current_page:number,  
8    last_page:number,  
9    per_page:number,  
10 }  
11  
12 export interface Response<T>{  
13   data:T  
14 }
```

Ilustración 32: Tipos de datos “*response.ts*”

Fuente: Elaboración propia

3.6.4 Uso de genérico

El genérico <T> permite declarar de forma explícita qué modelo se espera en cada solicitud, por ejemplo:

```
Promise<ResponsePagination<Observation>>=>
```

Ilustración 33: Ejemplo de uso para genéricos

Fuente: Elaboración propia

De esta forma, el sistema mantiene un tipado consistente sin duplicar estructuras para cada módulo.

3.6.5 Tabla de solicitudes HTTP del sistema

A continuación, se presenta la tabla de solicitudes HTTP del sistema, la cual describe los principales *endpoints* disponibles, lo métodos HTTP utilizados y su función dentro de la aplicación.

Módulo	Método	Endpoint	Descripción
Auth	Post	/auth/login	Autenticar usuario y obtener token
Auth	Post	/auth/logout	Cierra sesión del usuario
Auth	Get	/auth/me	Obtiene la información del usuario actualmente autenticado
User	Get	/users?page=\${page}	Lista todos los usuarios Parámetro: página
User	Get	/roles	Lista todos los roles disponibles
User	Post	/users/assign-roles ?user_id=\${id}&roles=\${roles}	Asigna rol a un usuario Parámetros: ID del usuario y los roles asignados
Template	Get	/template-forms/\${id}	Obtiene una plantilla específica por su ID
Template	Put	/template-forms/\${id}	Actualiza una plantilla por su ID
Template	Post	/template-forms	Crea una nueva plantilla
Template	Get	/template-forms?page=\${page}	Lista todas las plantillas con paginación Parámetro: página
Template	Delete	/template-forms/\${id}	Eliminar una plantilla
Template	Get	/template-forms/without-paginate	Lista todas las plantillas sin paginación
Observation	Get	/observations/\${id}	Obtiene una observación específica por su ID
Observation	Get	/observations/\${id}/form-with-answers	Obtiene una observación con su formulario y respuestas asociadas
Observation	Get	/observations?PRC_id=\${PRCId}\$page=\${page}	Lista todas las observaciones con paginación Parámetros: Id del PRC y página
Observation	Delete	/observation/\${id}	Elimina una observación específica
PRC	Get	/program-registers/\${id}	Obtiene un registro de programa específico

PRC	Get	/program-registers?page=\${page}	Lista registros de programa con paginación. Parámetro: página
PRC	Delete	/program-registers/\${id}	Elimina un registro de un programa específico
PRC	Patch	/program-registers/\${id}/change-status	Cambia el estado de un PRC
PRC	Get	/program-registers/catalog	Obtiene el listado de catalogo

Tabla 1: Tabla de solicitudes HTTP del sistema**Fuente:** Elaboración propia

3.7 Diseño de la interfaz

En este proyecto, el diseño de la interfaz de usuario (UI) se desarrolló con un enfoque centrado en la simplicidad, la usabilidad y la adaptabilidad, con el objetivo de ofrecer una experiencia de usuario (UX) óptima. Inicialmente, se realizaron encuestas internas en las que cada integrante del equipo votó sobre los elementos de diseño que resultaban más atractivos o funcionales, asegurando un consenso sobre la dirección del diseño. Posteriormente, se realizó la consolidación y materialización del diseño final, aplicando las decisiones tomadas de manera colectiva y utilizando herramientas y metodologías que garantizaran interfaces claras y funcionales antes de su implementación.

3.7.1 Estructura y diseño de la interfaz de usuario (UI)

La interfaz de usuario (UI) de este proyecto se diseñó con un enfoque modular y coherente, priorizando la usabilidad, la claridad y adaptabilidad. Cada módulo del sistema fue conceptualizado para cumplir objetivos específicos y facilitar la interacción del usuario con las funcionalidades disponibles, asegurando una experiencia de usuario (UX) óptima y consistente. Por ejemplo:

- **Autenticación (Auth)**

Incluye los formularios de inicio. Se priorizo la simplicidad y claridad, mostrando los campos y botones necesarios para completar la acción de manera rápida y segura.

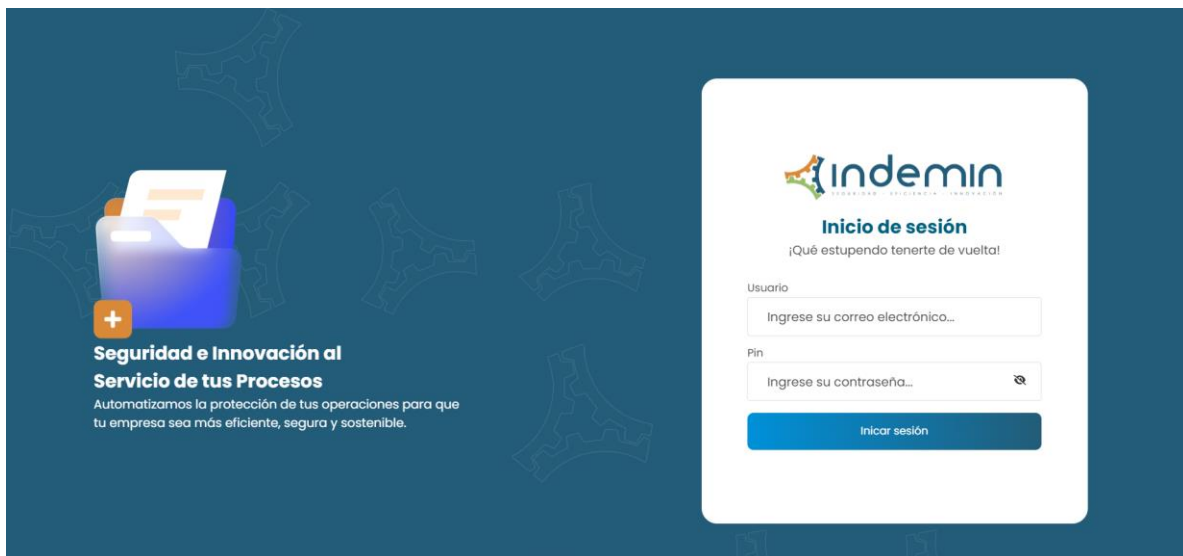


Ilustración 34: Vista Login
Fuente: Elaboración propia

- **Gestión de usuarios y roles (Users)**

Este módulo permite visualizar, administrar y asignar roles a los usuarios del sistema. Se diseñó una tabla (común para todas las vistas) y paginación para facilitar la navegación, además de contar con un modal que permite la edición del rol.

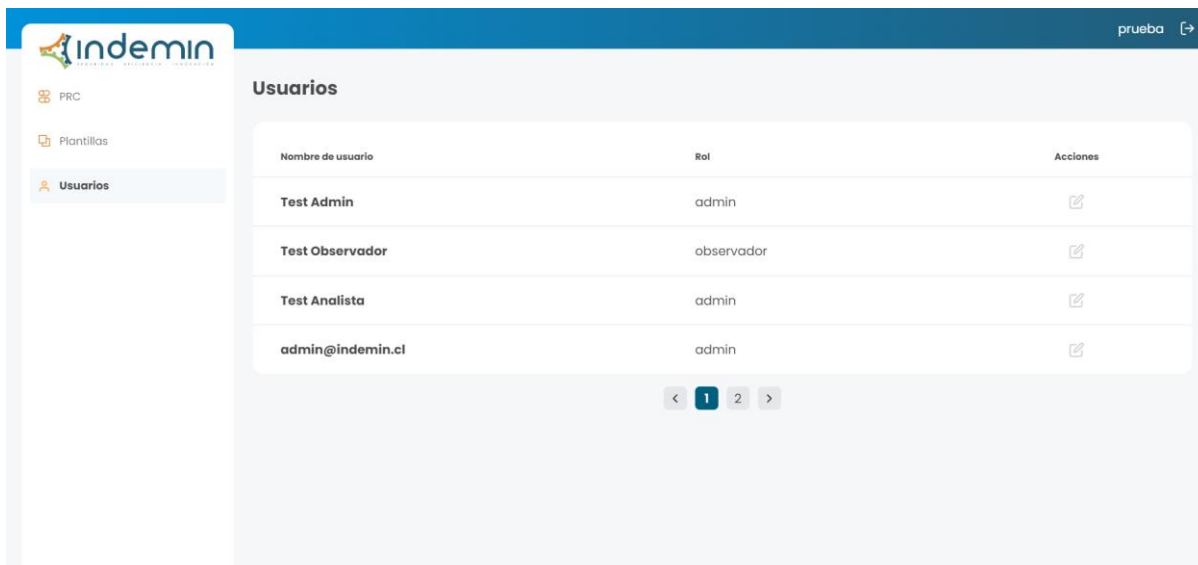


Ilustración 35: Vista listado de Usuarios

Fuente: Elaboración propia

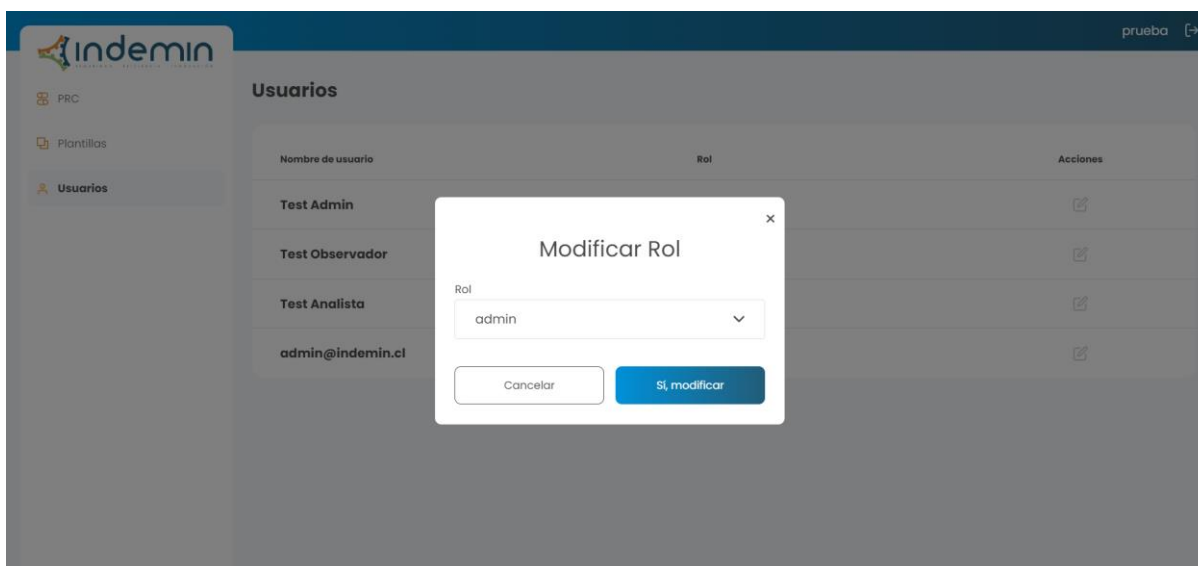


Ilustración 36: Vista modificar rol de Usuarios

Fuente: Elaboración propia

- **Plantillas (Templates)**

Permite crear, actualizar, visualizar y eliminar plantillas. La interfaz presenta una tabla con paginación que facilita la gestión de múltiples registros. Además, incluye una vista detallada de cada plantilla y un formulario para la creación o edición, donde se deben completar todos los campos requeridos. El sistema proporciona mensajes de error, ofreciendo un *feedback* visual claro al usuario en caso de que produzcan errores.

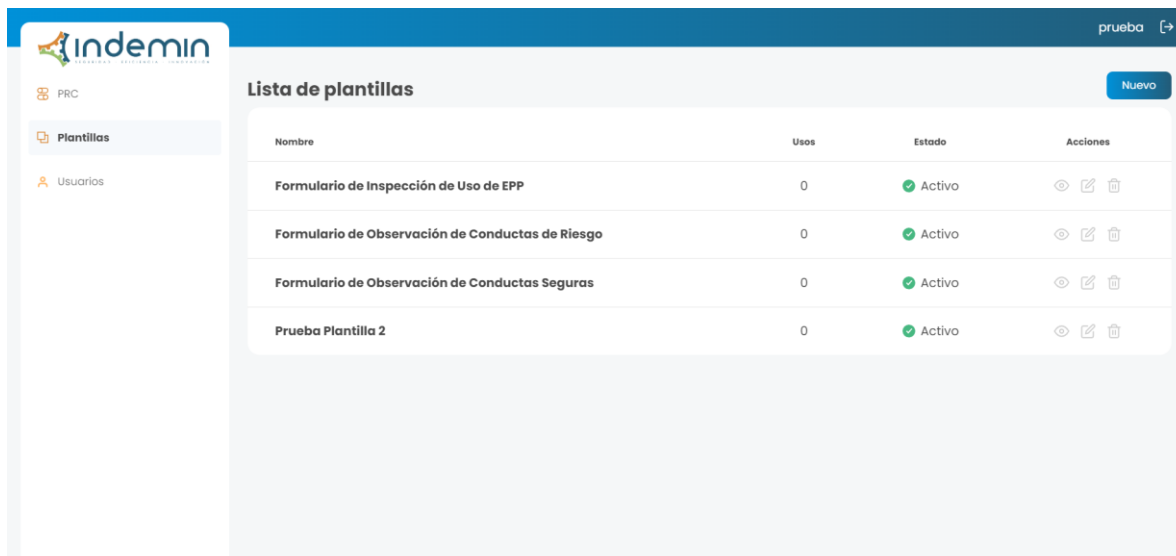


Ilustración 37: Vista listado de Plantillas
Fuente: Elaboración propia

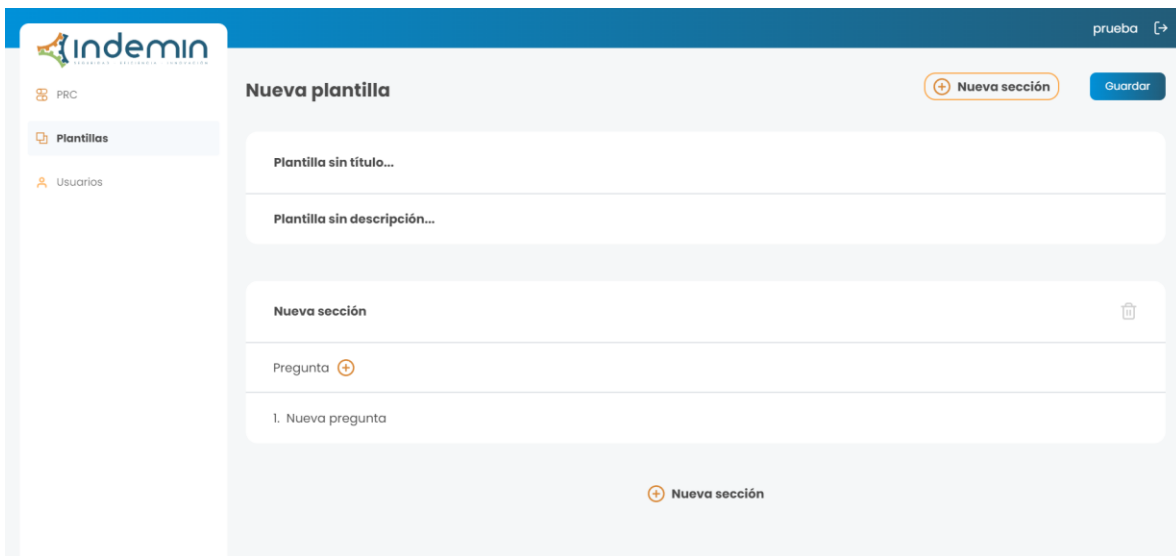
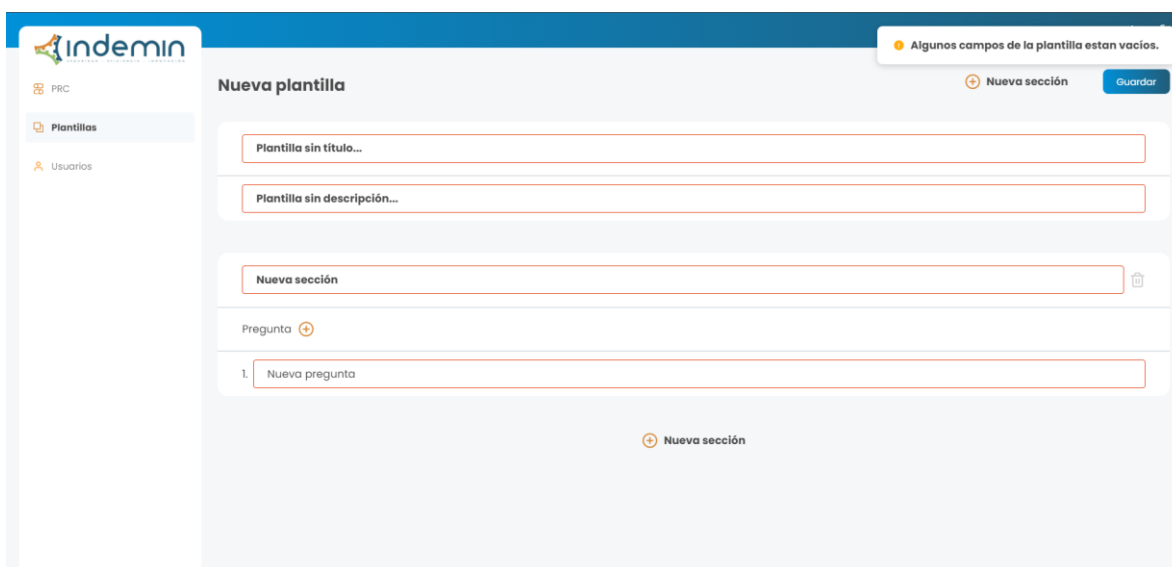


Ilustración 38: Vista formulario de Nueva Plantilla
Fuente: Elaboración propia



Indemin

PRC

Plantillas

Usuarios

Nueva plantilla

Algunos campos de la plantilla estan vacios.

Nueva sección Guardar

Plantilla sin título...

Plantilla sin descripción...

Nueva sección

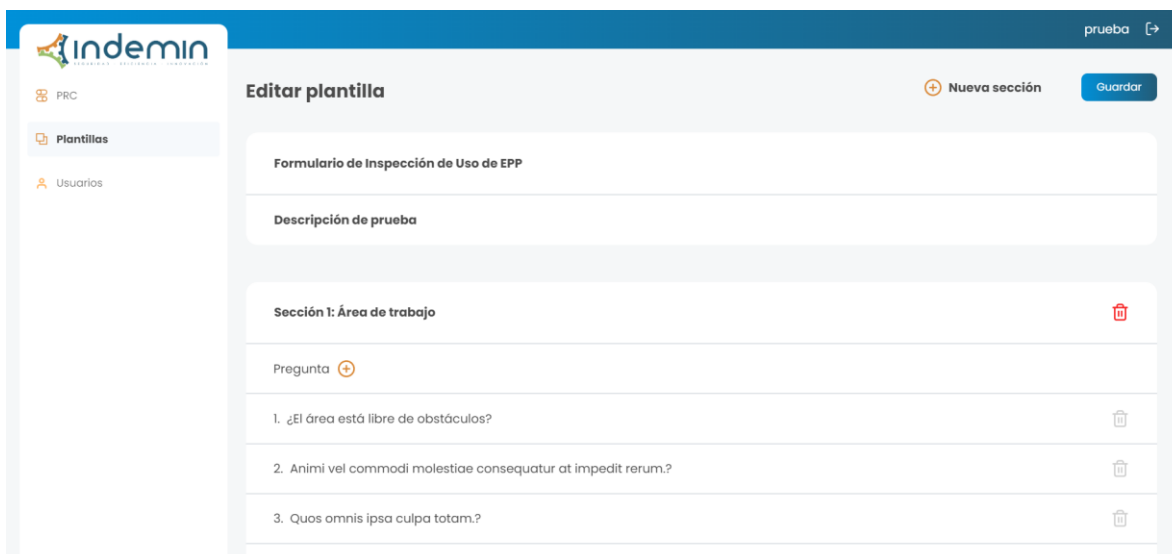
Pregunta

1. Nueva pregunta

Nueva sección

Ilustración 39: Vista formulario de Nueva Plantilla con errores

Fuente: Elaboración propia



Indemin

PRC

Plantillas

Usuarios

Editar plantilla

prueba [->]

Nueva sección Guardar

Formulario de Inspección de Uso de EPP

Descripción de prueba

Sección 1: Área de trabajo

Pregunta

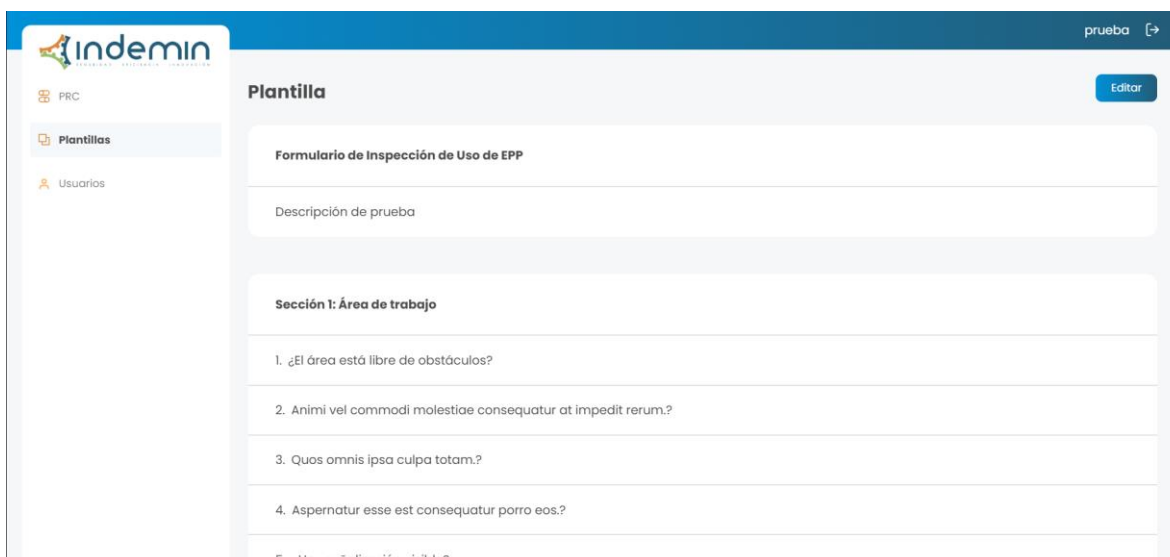
1. ¿El área está libre de obstáculos?

2. Animi vel commodi molestiae consequatur at impedit rerum.?

3. Quos omnis ipsa culpa totam.?

Ilustración 40: Vista formulario de Editar Plantilla

Fuente: Elaboración propia



Plantilla

Formulario de Inspección de Uso de EPP

Descripción de prueba

Sección 1: Área de trabajo

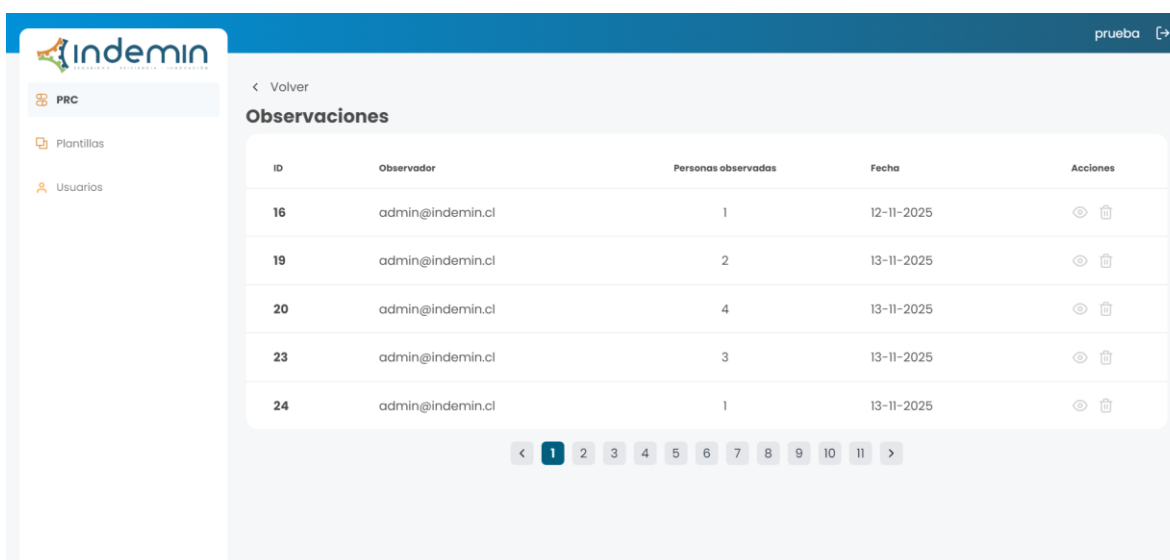
1. ¿El área está libre de obstáculos?
2. Animi vel commodi molestiae consequatur at impedit rerum.?
3. Quos omnis ipsa culpa totam.?
4. Aspernatur esse est consequatur porro eos.?
5. ¿Hay señalización visible?

Ilustración 41: Vista formulario de Editar Plantilla

Fuente: Elaboración propia

- **Observaciones (*Observations*)**

Permite visualizar todas las observaciones asociadas a un PRC, además de tener una interfaz específica para visualizar sus detalles y las respuestas al formulario.



Observaciones

ID	Observador	Personas observadas	Fecha	Acciones
16	admin@indemin.cl	1	12-11-2025	 
19	admin@indemin.cl	2	13-11-2025	 
20	admin@indemin.cl	4	13-11-2025	 
23	admin@indemin.cl	3	13-11-2025	 
24	admin@indemin.cl	1	13-11-2025	 

< 1 2 3 4 5 6 7 8 9 10 11 >

Ilustración 42: Vista listado de Observaciones

Fuente: Elaboración propia

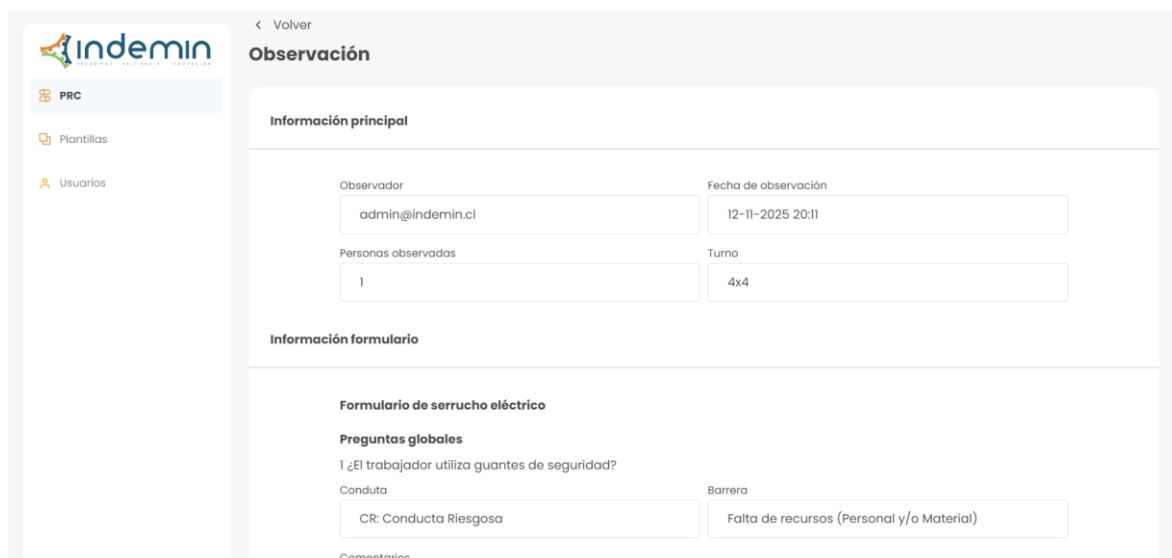


Ilustración 43: Vista detalle Observación
Fuente: Elaboración propia

- **Programas de registros conductuales (PRC)**

Permite consultar, actualizar, visualizar y eliminar programa de registros conductuales. La pantalla principal presenta una tabla con paginación. Adicionalmente, se incluye una vista detallada de cada registro, así como un formulario para la creación de nuevos programas, donde se deben completar los campos requeridos. El sistema también proporciona mensajes de error, ofreciendo un *feedback* visual claro al usuario sobre la correcta ejecución de las acciones o posibles incidencias.



ID	Programa registro conductual	Fecha inicio	Estado	Acciones
2	Operación maquinaria pesada	06-11-2025	En progreso	   
3	Formulario andamio	12-11-2025	En progreso	   
4	Formulario de tendido eléctrico	12-11-2025	En progreso	   
5	Formulario de serrucho eléctrico	12-11-2025	En progreso	   

Ilustración 44: Vista listado de Programas de Registros Conductuales
Fuente: Elaboración propia



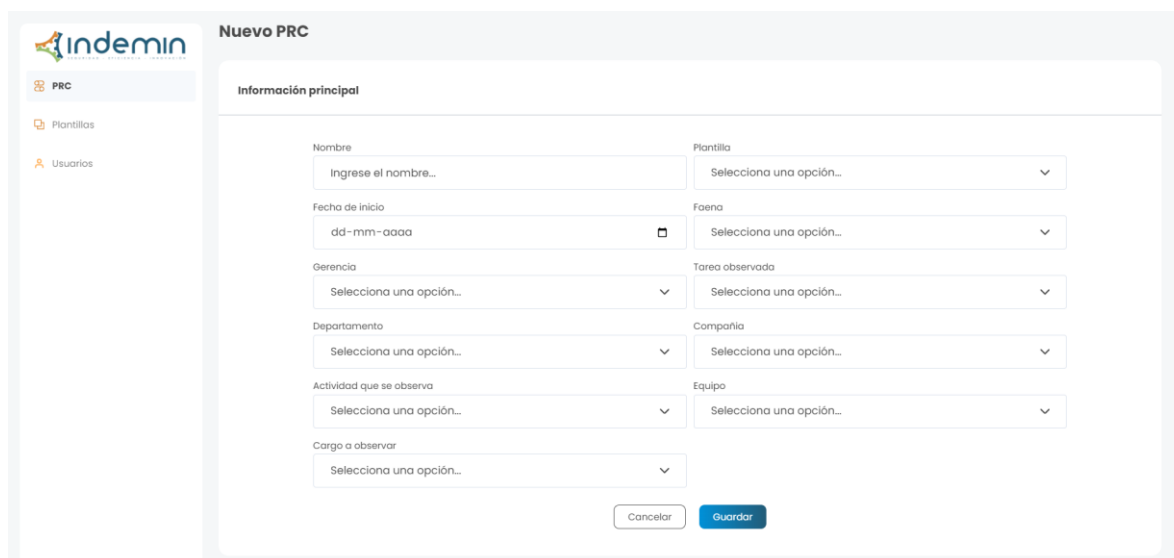
Programa de registro conductual

Observaciones

Información principal

Nombre	Plantilla
Operación maquinaria pesada	Operación de equipos
Fecha de inicio	Faena
06-11-2025	CUCONS-Ventanas
Gerencia	Tarea observada
Operaciones	Rutinaria
Departamento	Compañía
Prevención de riesgos	Codelco
Actividad que se observa	Equipo
Capacitación	Cargadores frontales
Cargo a observar	
Operador	

Ilustración 45: Vista detalle de un Programa de Registro Conductual
Fuente: Elaboración propia



Nuevo PRC

Plantillas

Información principal

Nombre	Plantilla
Ingrese el nombre...	Selecciona una opción...
Fecha de inicio	Faena
dd-mm-aaaa	Selecciona una opción...
Gerencia	Tarea observada
Selecciona una opción...	Selecciona una opción...
Departamento	Compañía
Selecciona una opción...	Selecciona una opción...
Actividad que se observa	Equipo
Selecciona una opción...	Selecciona una opción...
Cargo a observar	
Selecciona una opción...	

Cancelar Guardar

Ilustración 46: Vista formulario de crear Programa de Registro Conductual
Fuente: Elaboración propia

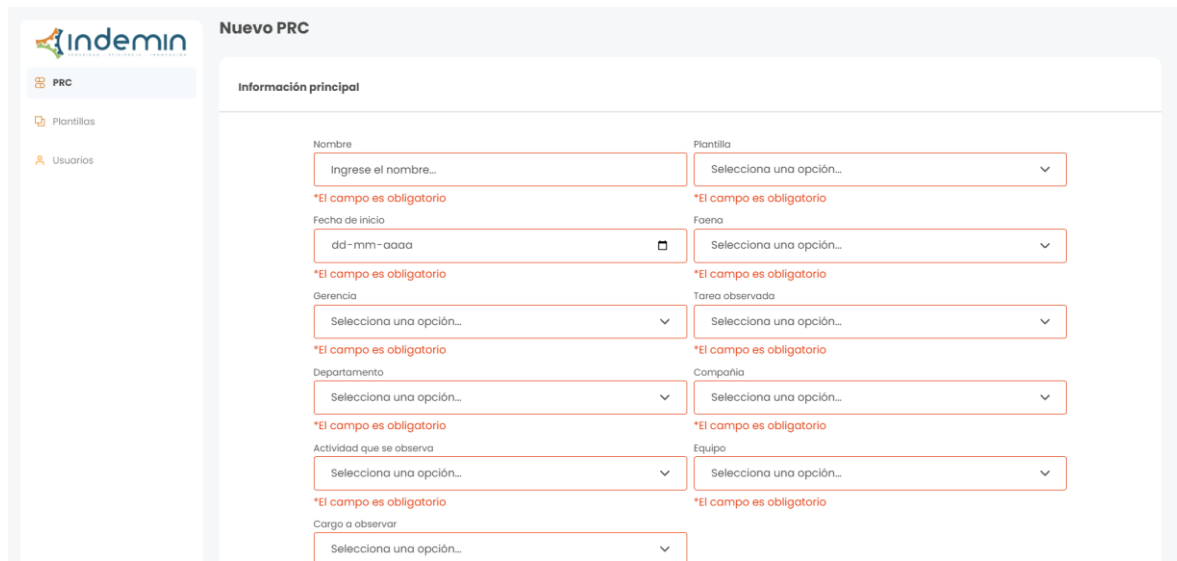


Ilustración 47: Vista formulario de crear Programa de Registro Conductual
Fuente: Elaboración propia

3.8 Detalles de la codificación y desarrollo

3.8.1 Backlog General del proyecto

El siguiente backlog corresponde únicamente a las actividades vinculadas con el desarrollo web del proyecto, abarcando desde la planificación inicial hasta la implementación funcional de cada uno de los módulos del sistema. Su objetivo es describir de manera ordenada y progresiva las tareas realizadas en cada etapa.

- **Inicio Proyecto**
 - Definir de los objetivos principales del sistema web.
 - Recopilación y análisis de la información necesaria para comprender los requerimientos funcionales.
 - Elaboración y validación inicial de historias de usuario, orientadas exclusivamente al flujo del sistema web.
 - Diseño de mockups.
 - Selección de librerías, *frameworks* y herramientas.
- **Sprint 0**
 - Presentación y validación de la propuesta inicial del sistema web
 - Refinamiento y confirmación de las historias de usuario pertinentes al desarrollo *front-end*.
- **Sprint 1**
 - Configuración inicial del entorno: creación del proyecto en Vue 3, estructura base de carpetas y herramientas esenciales.
 - Configuración inicial de control de versiones (Git) y lineamientos de trabajo.

- **Sprint 2**
 - Definición y organización de la arquitectura base del proyecto web.
 - Configuración del sistema de rutas (Vue Router), creando navegaciones iniciales entre vistas principales.
- **Sprint 3**
 - Implementación de la funcionalidad de autenticación (login).
 - Creación de la vista inicial de "Plantillas".
 - Desarrollo de la funcionalidad completa del módulo "Plantillas", incluyendo tabla, detalle y formulario.
- **Sprint 4**
 - Diseño y construcción de la vista correspondiente al módulo "PRC".
- **Sprint 5**
 - Implementación de las funcionalidades asociadas a la vista "PRC", contemplando tabla, detalle y creación.
- **Sprint 6**
 - Generación de las vistas base del módulo "Observación".
 - Implementación completa del flujo de observaciones: tabla paginada, vista detallada y formulario.
- **Sprint 7**
 - Implementación de validaciones de formularios en distintos módulos, incluyendo manejo de errores y mensajes de retroalimentación visual.
- **Sprint 8**
 - Creación de la vista "Usuarios".
 - Implementación de la funcionalidad de asignación y actualización de roles de usuario.
 - Incorporación de mejoras a la autenticación, incluyendo login (Indemin) y manejo avanzado de la sesión (permitir solo usuarios con rol administrador).

3.8.2 Control de versiones

Para la gestión de código fuente se utilizó Git como sistema de control de versiones. Su uso permitió mantener un historial claro de cambios, facilitar el trabajo iterativo por módulos y asegurar una trazabilidad completa durante el desarrollo.

El proyecto siguió las convenciones de Git Flow, organizando el trabajo en ramas específicas según su propósito:

- Main para la versión estable.
- Develop como rama de desarrollo y qa.
- Ramas de feature para el desarrollo de funcionalidades
- Ramas de fix para resolver conflictos o errores

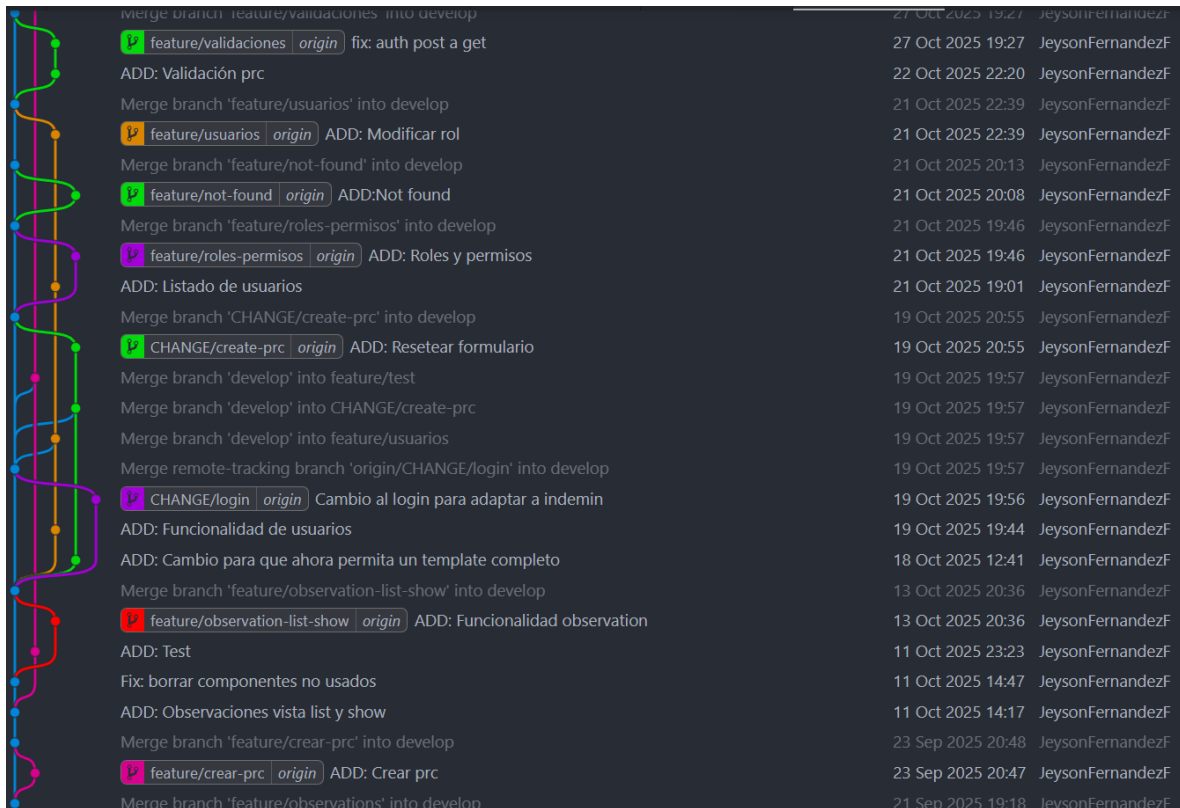


Ilustración 48: Historial de GitHub

Fuente: Elaboración propia

3.9 Pruebas y validación

La validación del sistema se realizó para asegurar que todas las funcionalidades cumplieran con los requerimientos planteados y que la experiencia de usuario fuera adecuada. Para ello, se llevaron a cabo tres tipos de actividades complementarias:

1. Pruebas unitarias, enfocadas en verificar que cada componente y función del *front-end* operara correctamente de manera independiente.
2. Pruebas *end-to-end* (E2E), destinadas a evaluar el flujo completo del sistema, desde la autenticación hasta la gestión de los datos en diferentes módulos, asegurando la integración adecuada de todas las funcionalidades.
3. Visita a terreno, en la que se observó la interacción de los usuarios finales con la interfaz, permitiendo identificar aspectos de usabilidad y eficiencia en condiciones reales de uso.

A continuación, se describen en detalle cada uno de estos enfoques de validación.

3.9.1 Pruebas unitarias

Las pruebas unitarias se realizaron utilizando Vitest, enfocándose en los componentes y funciones del sistema web. Estas pruebas tienen como objetivo verificar el correcto

funcionamiento de cada unidad del código de manera aislada, asegurando que cada módulo cumpla con su propósito antes de integrarse con el resto del sistema.

Entre los principales beneficios de las pruebas unitarias se encuentran:

- Verificar que cada módulo funcione según lo esperado de manera aislada.
- Detectar errores tempranamente en la lógica de formularios, tablas y funcionalidades específicas.
- Garantizar que los cambios futuros en el código no afecten funcionalidades existentes, facilitando la refactorización y el mantenimiento del sistema.

```
DEV v3.2.4 C:/Users/jexso/Desktop/Proyecto Universidad/frontapicoresafe

stdout | src/test/user/useUsers.test.ts > useUsers composable > getUsers carga usuarios y actualiza store y paginación
false

stdout | src/test/observation/useObservationForm.test.ts > useObservationForm composable > getFormByObservation exitoso
{ id: 1, name: 'Formulario Test' }

stdout | src/test/user/useUsers.test.ts > useUsers composable > getUsers lanza error si falla la API
false

✓ src/test/observation/useObservationForm.test.ts (3 tests) 7ms
✓ src/test/user/useUsers.test.ts (6 tests) 10ms
stdout | src/test/observation/useObservation.test.ts > useObservation composable > getObservation carga la observación y actualiza la store
{ data: { id: 1, title: 'Test Observation' } }

✓ src/test/observation/useObservation.test.ts (3 tests) 8ms
✓ src/test/template/useTemplates.test.ts (6 tests) 8ms
✓ src/test/prc/composables/usePrcs.test.ts (2 tests) 12ms
stdout | src/test/prc/views/PrcView.test.ts > PrcView.vue > renders the component when PRC exists
123

stdout | src/test/prc/views/PrcView.test.ts > PrcView.vue > shows loader when PRC does not exist
123

✓ src/test/prc/views/PrcView.test.ts (2 tests) 31ms
✓ src/test/prc/composables/usePrc.test.ts (3 tests) 7ms
✓ src/test/observation/useObservationDelete.test.ts (3 tests) 7ms
✓ src/test/user/useChangeRole.test.ts (4 tests) 8ms
✓ src/test/template/useTemplate.test.ts (3 tests) 6ms
✓ src/test/observation/useObservations.test.ts (4 tests) 7ms

Test Files 11 passed (11)
Tests 39 passed (39)
Start at 17:01:05
Duration 3.67s (transform 604ms, setup 0ms, collect 2.58s, tests 111ms, environment 13.02s, prepare 2.07s)
```

Ilustración 49: Visión global de las pruebas unitarias realizadas

Fuente: Elaboración propia

Los resultados mostraron que todas las funciones críticas operaban correctamente y que los componentes respondían de manera estable a distintos casos de prueba, contribuyendo a la calidad y confiabilidad del sistema.

3.9.2 Pruebas End-to-End (E2E)

Las pruebas *end-to-end* (E2E) se realizaron utilizando Playwright, enfocándose en validar el flujo completo de interacción del usuario dentro del sistema web. Estas pruebas tienen como objetivo comprobar que todos los componentes y módulos del sistema funcionen correctamente cuando se integran, asegurando que las tareas de usuario puedan completarse de manera exitosa de principio a fin.

- Validar la navegación y la interacción entre módulos, incluyendo login, plantillas, programas de registro (PRC), observaciones y usuarios.

- Asegura que los formularios, tablas y notificaciones de *feedback* visual funcionen correctamente.
- Detectar errores o inconsistencias en la integración de los componentes, que no podrían identificarse mediante pruebas unitarias.

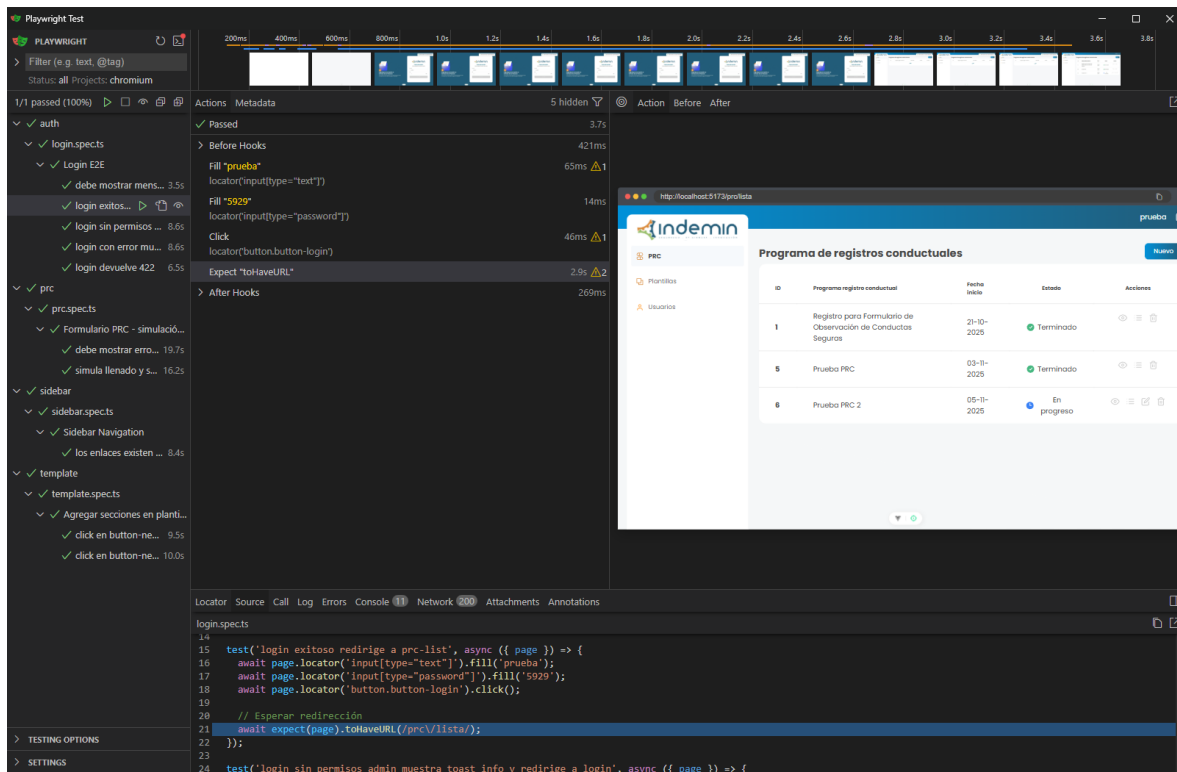


Ilustración 50: Visión global de las pruebas End-To-End realizadas
Fuente: Elaboración propia

Los resultados obtenidos confirmaron que los flujos principales del sistema operaban de manera estable, que la interacción entre módulos era coherente y que las tareas críticas podían realizarse sin inconvenientes.

3.9.3 Visita a terreno

Como parte del proceso de validación, se realizó una visita a terreno para evaluar la interacción de los usuarios finales con el sistema web en condiciones reales. El objetivo de esta actividad fue observar la usabilidad, eficiencia y satisfacción de los usuarios, así como recopilar retroalimentación directa que permitiera mejorar la experiencia de uso. Durante la visita, los usuarios destacaron varios aspectos positivos del sistema:

- Integración con los sistemas existentes de Indemin, lo que les permite iniciar sesión con los mismos usuarios que utilizan en otras plataformas, simplificando la gestión de credenciales.



- La visualización de las observaciones realizadas les resultó clara y organizada, facilitando el seguimiento de los registros.
- La creación y reutilización de plantillas, que les permitió sentirse cómodos y eficientes al generar nuevos usuarios, permitiendo generar Programas de Registros Conductuales de formas más fácil y específicos.

Asimismo, los usuarios sugirieron la inclusión de un botón en la parte inferior del formulario para agregar más secciones, lo que agiliza la creación de plantillas más complejas. Esta funcionalidad ya fue incorporada al sistema, respondiendo directamente a sus necesidades y mejorando la experiencia de uso.

4 Conclusión

Este proyecto significó un avance importante en la modernización de los procesos internos de la organización, reemplazando un sistema basado en formulario de papel por una plataforma digital estructurada, segura y escalable. La implementación de esta solución permitió optimizar la gestión de programas de registros conductuales, plantillas dinámicas y observaciones en terreno, mejorando no solo la eficiencia operativa, sino también la calidad y trazabilidad de la información recopilada.

El uso de Vue 3 como *framework* principal proporcionó una base sólida para construir una interfaz dinámica, capaz de adaptarse a las necesidades de los usuarios que trabajan en la plataforma web. La arquitectura modular permitió mantener el sistema organizado, separando de forma clara cada área funcional (como PRC, plantillas, observaciones y usuarios) y permitiendo que cada módulo pudiera crecer o modificarse sin afectar al resto del sistema. Esta estructura también favoreció la reutilización de componentes y *composables*, mejorando la consistencia visual y reduciendo la duplicación de código.

Otro aspecto fue la incorporación de pruebas automatizada y evaluaciones en terreno, lo que permitió validar el comportamiento del sistema en contextos reales de uso. Estas pruebas se pudieron identificar mejoras antes de la puesta en producción, asegurando una experiencia estable y un funcionamiento adecuado en condiciones operativas. Además, el uso de herramientas de control de versiones, como Git, resultó esencial para mantener un registro claro de los cambios, coordinar correcciones y garantizar la trazabilidad del desarrollo.

En términos operativos, el sistema representa un cambio profundo respecto al trabajo tradicional basado en papel. La digitalización de los registros conductuales permitió mejorar la rapidez del proceso, eliminar errores asociados al manejo físico de documentos y facilitar el análisis posterior de la información. Durante las presentaciones en terreno, los usuarios destacaron que los plantillas son adaptables a todos los PRC, lo que facilita su uso en distintos contextos y asegura una mayor flexibilidad en la recolección de datos.

Durante el desarrollo del proyecto se identificaron varios aprendizajes clave. Primero, la importancia de involucrar a los usuarios finales desde etapas tempranas: haber conversado con ellos desde el inicio nos habría permitido entender mejor el negocio y evitar la necesidad de implementar ciertas mejoras posteriormente. Segundo, la relevancia de una arquitectura modular y de componentes reutilizables, que facilita el mantenimiento y la escalabilidad del sistema a largo plazo. Tercero, las pruebas en terreno y la documentación constante de los cambios demostraron ser fundamentales para anticipar problemas y asegurar una implementación exitosa. Finalmente, el trabajo en equipo fue complejo, pero mantener reuniones frecuentes ayudó a coordinar esfuerzos, mantener a todos alineados y mejorar la organización del proyecto.

Como proyecciones y mejoras a futuro, y en base a las conversaciones sostenidas con los encargados de Indemin, se identificaron nuevas funcionalidades que podrían fortalecer aún más la plataforma. Entre ellas, destaca la posibilidad de descargar reportes consolidados de los Programas de Registro Conductual y sus observaciones, lo que facilitaría el análisis externo y la toma de decisiones. Asimismo, se consideró relevante que se incorpore la opción de descargar los formularios completos en formato pdf, permitiendo que la organización cuente con respaldos formales para procesos de auditoría.



Finalmente, es importante señalar que el presente trabajo no incluye métricas cuantitativas, tales como mediciones de tiempos de respuesta, reducción objetiva de errores o comparativas de desempeño antes y después de la implementación. Esto se debe a que, al momento de desarrollar y presentar el proyecto CoreSafe el sistema no se encontraba en operación real, por lo que no fue posible recopilar datos asociados a su uso productivo.

En conjunto, el proyecto demuestra que una plataforma bien diseñada (basada en Vue, con arquitectura modular, pruebas sistemáticas y componentes reutilizables) puede generar mejoras sustanciales tanto en la gestión interna como en el desempeño operativo de quienes la utilizan. La transición desde los procesos manuales hacia una herramienta digital no solo moderniza la operación, sino que además refuerza la seguridad, eficiencia y calidad del trabajo realizado.

5 Referencias

- [1] Vue.js, "Documentación" [Online]. Disponible: <https://vuejs.org/guide/introduction.html>
- [2] Metodología agile. Disponible: <https://agilemanifesto.org/>
- [3] Vue Testing, "Documentación" [Online]. Disponible: <https://vuejs.org/guide/scaling-up/testing>
- [4] Vitest, "Documentación" [Online]. Disponible: <https://vitest.dev/>
- [5] Playwright, "Documentación" [Online]. Disponible: <https://playwright.dev/docs/intro>
- [6] JWT, "Definición" [Online]. Disponible: <https://www.ibm.com/docs/es/cics-ts/6.x?topic=cics-json-web-token-jwt>
- [7] E. Freeman and E. Robson, Head First JavaScript Programming. Sebastopol, CA, USA: O'Reilly Media, 2020.
- [8] D. Flanagan, JavaScript: The Definitive Guide, 7th ed. Sebastopol, CA, USA: O'Reilly Media, 2020
- [9] Robert C. Martin: The Clean Coder: A Code of Conduct for Professional Programmers
- [10] Composition Api, "Documentación" [Online]: <https://vuejs.org/guide/extras/composition-api-faq.html>
- [11] TypeScript, "Documentación" [Online]. Disponible: <https://www.typescriptlang.org/>
- [12] Pinia, "Documentación" [Online]. Disponible: <https://pinia.vuejs.org/>
- [13] Tailwindcss, "Documentación" [Online]. Disponible: <https://tailwindcss.com/>
- [14] Scss, "Documentación" [Online]. Disponible: <https://sass-lang.com/>
- [15] Fetch API, "Documentación" [Online]". Disponible: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API
- [16] Robert C. Martín: Clean Architecture: A Craftsman's Guide to Software Structure and Design